(4)

RADC-TR-88-132, Vol I of
Final Technical Report
June 1988

CRONUS, A DISTRIBUTED OPERATING SYSTEM: Revised System/Subsystem Specification

BBN Laboratories Incorporated

R. Schantz, K. Schroder, M. Barrow, G. Bono, M. Dean, R. Gurwitz, K. Lam, K. Lebowitz, S. Lipson, P. Neves, R. Sands and R. Thomas

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss AFB, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-132, Volume I (of four) has been reviewed and is approved for publication.

APPROVED: *Thomas F. Lawrence*

THOMAS F. LAWRENCE
Project Engineer

APPROVED: *Raymond P. Urtz Jr.*

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: *John A. Ritz*

JOHN A. RITZ
Directorate of Plans and Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific doucment require that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) Report No. 5884	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-132, Volume I (of four)
6a. NAME OF PERFORMING ORGANIZATION BBN Laboratories Incorporated 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)
6c. ADDRESS (City, State, and ZIP Code) 10 Moulton Street Cambridge MA 02238	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-84-C-0171
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: 63728F; PROJECT NO: 2530; TASK NO: 01; WORK UNIT ACCESSION NO: 26

11. TITLE (Include Security Classification)
CRONUS, A DISTRIBUTED OPERATING SYSTEM: Revised System/Subsystem Specification

12. PERSONAL AUTHOR(S) R. Schantz, K. Schroder, M. Barrow, G. Bono, M. Dean, R. Gurwitz, K. Lam, K. Lebowitz, S. Lipson, P. Neves, R. Sands and R. Thomas

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Oct 84 TO Jan 86	June 1988	176

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Distributed Operating System, Heterogeneous Distributed System, Interoperability, System Monitoring & Control, Survivable Application
12	07		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This is the final report for the second contract phase for development of the Cronus Project. Cronus is the name given to the distributed operating system (DOS) and system architecture for distributed application development environment being designed and implemented by BBN Laboratories for the Air Force Rome Air Development Center (RADC). The project was begun in 1981. The Cronus distributed operating system is intended to promote resource sharing among interconnected computer systems and manage the collection of resources which are shared. Its major purpose is to provide a coherent and integrated system based on clusters of interconnected heterogeneous computers to support the development and use of distributed applications. Distributed applications range from simple programs that merely require convenient reference to remote data, to collections of complex subsystems tailored to take advantage of a distributed architecture. One of the main contributions of Cronus is a unifying architecture and model for developing these distributed applications, as well as support for a number of system-provided functions which are common to many applications.

(Over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas F. Lawrence	22b. TELEPHONE (Include Area Code) (315)330-2158 22c. OFFICE SYMBOL RADC (COTD)

DD Form 1473, JUN 86 *Previous editions are obsolete*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Block 19 (Cont'd)

This report consists of four volumes:

Vol I - CRONUS, A DISTRIBUTED OPERATING SYSTEM: Revised System/Subsystem Specification

Vol II - CRONUS, A DISTRIBUTED OPERATING SYSTEM: Functional Definition and System Concept

Vol III - CRONUS, A DISTRIBUTED OPERATING SYSTEM: Interim Technical Report No. 5

Vol IV - CRONUS, A DISTRIBUTED OPERATING SYSTEM: CRONUS DOS Implementation

Table of Contents

FIGURES

TABLES

1. Introduction

This report presents the current design for Cronus, the system being developed under the Distributed Operating System Design and Implementation project sponsored by Rome Air Development Center[1]. It is intended as an overview of the system structure and as a synopsis of the current system subsystem decomposition and design. A previous report, *Cronus, A Distributed Operating System, Functional Definition and System Concept*, BBN Report No. 5879, is intended as a companion to the current report, and the reader is assumed to be familiar with its contents.

The first three editions of this specification were produced under the previous contract: two as part of interim technical reports and the third as an independent document. These early revisions served to formalize our notions of how a geographically distributed, heterogeneous system built from interconnected processing systems should be organized. Preliminary implementations of many system components were produced to confirm the viability of our approach, but little experience with component interactions or with use of the services by clients occured during that early period.

This and the previous edition reflect the fact that most of what we originally described has now been implemented and has been put to practical use: kernel functions such as interprocess communication and initial versions of system services such as host management, process management, catalogs, files and access control have been completed, have experienced substantial use and have become quite stable at this point. In a few areas, such as device support and user interfaces, we have not yet had substantial exprience. In these areas, we have relied upon the services provided by the constituent operating systems of the hosts to provide functions such as tape archival, terminal input and interactive command execution. For these areas, this report briefly describes the extent of the current implementations and presents ideas about how the service might be better supported after further development.

This edition includes a new section discussing tools for distributed application development. From our experience in building system managers, we have introduced tools to formalize and automate many aspects of the development process. We now regularly use these tools to produce new application components.

In Section 2, we briefly review a few of the areas covered in the Functional Definition, and extend them to cover current development plans.

Section 3 presents an overview of the Cronus operating system, stressing the common framework into which its components will fit and the functional decomposition of the system.

[1]This work has been performed under RADC contracts F30602-84-C-0171 and F30603-81-C-0132.

Sections 4 through 13 present the design for the various system functions. An initial implementation has been provided in most of these areas. Our experience in using these components varies from the kernel and other system functions, which were provided early, to devices and user interfaces, where our implementation is most limited. These sections will form the basis of a continuing and evolving subsystem specification for the various components, throughout the life of the project.

The remaining sections describe the system environment. Section 14 describes the hardware that supports the current Cronus implementation. Section 15 describes the functions required of an underlying network. Section 16 describes how special capabilities common to local area networks, such a broadcast messages service, are provided when the underlying network consists of multiple local area networks connected by gateways or other networks. Section 17 describes the facilities of the generic computing element.

2. Cronus Project Overview

2.1. Project Objectives

The objective of the Cronus project is to develop a testbed for evaluating distributed system technology. To do this we are establishing a prototype local area network based hardware architecture, and building an operating system and software architecture to organize and control this distributed system. The architecture is described in the Cronus Functional Description [BBN 5879], and is summarized in Section 4. In addition to establishing a system architecture, the other major aspects of the Cronus project activities are:

1. *Select* off-the-shelf hardware and software components as a basis for an Advanced Development Model (ADM) prototype configuration for the distributed system testbed.
2. *Design* the system.
3. *Implement* a version of the basic system components.
4. *Test* and evaluate the concepts and realization of the DOS in the Advanced Development Model.

The orientation we have chosen is both experimental through construction of working system components, and evolutionary through pre-planned continuation of design and development activities.

2.2. Points of Emphasis

The Cronus design is intended to introduce coherence and uniformity to a set of otherwise independent and disjoint computer systems. This grouping of machines, operating under the control of a distributed operating system, is called a Cronus cluster. The aim is to provide for the cluster configuration as a whole, features comparable to those found in a modern centralized computer utility. There are various ways of viewing this uniformity and coherence; each plays a role in the Cronus design.

From an end user's point of view, the Cronus DOS provides a single account with controlled access to all integrated system services in a manner which is independent of the site of the activity. From a programmer's point of view, Cronus supports a distributed programming paradigm which provides a uniform interface and access path to the distributed system resources, and supports the initiation and control of distributed computations. More importantly, from both an end user's and programmer's perspective, Cronus provides a *common system framework* for applications. This means that otherwise independent computerized activities can be constructed so that they are more easily made to work together, despite implementations which cross host and processor-type boundaries.

From an operations and administrative perspective Cronus provides a logically centralized facility for monitoring and controlling all of the connected systems. Functions such as account authorization, user priority, and access control can be applied system-wide rather than individually to each host.

In addition to coherence and uniformity, there are a number of other system design goals. These are:

- Survivability and integrity of Cronus itself and of some of the applications that use Cronus;
- Scalability to accommodate both small and large configurations and to support incremental growth;
- Experimentation with resource management strategies that effect global performance;
- Component substitutability to allow easy use of alternate functionally equivalent hardware and software support components; and
- Convenient operation and maintenance procedures.

2.3. The Cronus Hardware Architecture

2.3.1. System Environment

The Cronus environment consists of several parts: a set of local area networks that provide the communications substrate for a Cronus cluster, the set of hosts upon which the Cronus system operates, and a mechanism for connecting a Cronus cluster to the Internet environment and to other Cronus clusters.

Cronus enables a variety of constituent computer systems to operate in an integrated manner. Cronus is distinguished from other distributed operating systems by one or more of the following characteristics:

1. Cronus will most often run on a group of heterogeneous hosts. Cronus is oriented toward quickly enabling developers to gain access to and exploit the unique qualities of resources in a heterogeneous environment and providing a coherent model for such integrated heterogeneous systems.
2. The Cronus distributed operating system software often runs as an adjunct to, rather than a replacement for the hosts' primary operating systems. In these cases the original hosts operating system runs largely unmodified. Also under development is a version of Cronus as a base level operation system.
3. Hosts will be included in Cronus with varying degrees of system integration. Some support limited subsets of the services defined by the Cronus environment.
4. The interconnection network is designed on a hierarchical model. A Cronus cluster includes a set of hosts connected by a high-speed, low-latency local network. A set of Cronus clusters may be connected over slower long-haul networks.

The Cronus architecture provides a flexible environment for connecting hosts so that facilities available on one host may be conveniently used from other hosts. It provides two alternative host integration schemes. A host may implement the Cronus Interprocess Communication (IPC) mechanism and have efficient communication and operations with the rest of the Cronus hosts; or it may access the other Cronus hosts through a front end access machine, which is a simpler, less expensive option for connection of a host, but which may be more limited from a flexibility and performance viewpoint.

2.3.2. Host Classes

Cronus hosts can be divided into four groups: mainframe hosts, Generic Computing Elements (GCEs), workstations, and internet gateways.

The collection of mainframe hosts, each of which serves a number of users simultaneously, includes a variety of machines with unrelated architectures. A mainframe host may be tightly integrated into the system, both offering and using Cronus services and fully implementing Cronus interprocess communication. Alternatively, they may be loosely integrated offering no services, possibly connecting into Cronus through an access machine which provides communication with the rest of Cronus.

GCEs are small, dedicated-function microprocessor based computers of a single architecture but varying configuration. Each GCE provides a basic service. For example, a GCE can be a file manager, a terminal manager, an acccess machine or it might carry out a more complex system function as an authorization manager. Since all GCEs have the same architecture, they provide a replicated resource which, with the appropriate software, enhances the reliability of basic Cronus functions.

Workstations are powerful, dedicated computers which provide substantial computing power and graphics capability at the disposal of a single user. They differ from mainframes in that they support a single user. They differ from terminals in that they offer significant computational resources.

An internet gateway is a computer used to interface communication between multiple networks. The Cronus gateway integrates the Cronus cluster into the collection of networks known as the ARPA Internet and provides a base for supporting remote access and intercluster communication.

2.3.3. System Access

There are a variety of use access paths to Cronus. One is a connection by means of a Cronus terminal concentrator. Users may gain access through the internet gateway from remote points. Cronus also supports access through terminal access mechanisms on its mainframe hosts. These latter two access paths provide the same interface to the user as the terminal concentrator. Access from a workstation may be different than from a terminal, since the workstation defines the user interface. The user has immediate access to the workstation's capabilities.

2.3.4. Local Area Network

The set of hosts is connected by a local area network. The characteristics of the network play an important role in the design of Cronus applications, since they determine the kinds of communication and operations that are feasible across host components of Cronus.

The selection of an Ethernet for the local area network for the Advanced Development Model has been described in BBN Report 5086. This choice was motivated by criteria in the project's original statement of work:

1. The network should be suitable to support a distributed operating system,
2. The network should be currently available and economical. Since the Advanced Development Model will not be operated in a stressed environment, certain constraints applicable to a field-deployable version were considerably relaxed.

The Ethernet was chosen for the local area network substrate for the following reasons:

- It is desirable, though not required, that the network be "high-speed". The Ethernet operates at 10 MBits.
- Network interfaces to all or most of the computer systems in the DOS ADM should be available.
- The local network must provide a datagram-style service.

The Ethernet fulfills all three requirements and we believe is, at the present time, the most cost-effective network technology which does. In addition, the Ethernet provides broadcast and multicast capabilities which, have been extensively exploited in the system design.

The raw Ethernet layer is not used directly. To achieve convenient substitutability of alternate communication substrates, Cronus uses an abstraction of the Ethernet capabilities which is provided by a Virtual Local Net (VLN) software layer, described in Section 14.2. The VLN represents an enhancement of the DOD standard IP protocol to provide for features common to local area communication. We anticipate that future versions of Cronus will need to be built upon a different local network, such as the Flexible Flexible Interconnect, which have reliability, communication security, and ruggedization not available in current commercial products. By designing the VLN layer and building Cronus upon it, it should be easy to substitute any local network that provides the basic transport services required by Cronus.

This design is being extended to include clusters connected by a heterogeneous network layer, as when multiple Local Area Networks (LANs) are connected via gateways and the Arpanet. The features provided by the LAN may be used directly for communications between components on the same LAN. Features not supported by some of the networks in the network layer are provided by adding software to the gateways or hosts on the networks. For example, a broadcast repeater is used to propagate broadcast requests between interconnected LANs. Note that additional performance considerations may arise when dealing with heterogeneous networks. In particular, the bandwidth for messages passing through a

gateway or over land lines is typically poorer than the bandwidth of connections between hosts connected to the same local area network.

2.3.5. Types of Hosts

GCEs are implemented in the ADM system by Multibus computers with an MC68000 processor, large main memories, an Ethernet controller, and additional hardware (disks, RS-232 ports, etc) needed to support specific functions[2]. The Multibus computers were chosen because

1. They are relatively inexpensive, permitting low cost incremental system growth.
2. The Multibus standard guarantees the ability to package individual GCEs in different ways with components from a variety of vendors.
3. New processors and devices are expected to evolve for the Multibus over time.

Utility hosts provide the program development and application execution environments for Cronus. In the ADM, this function is supported by C70 UNIX systems, VAX-UNIX Systems and a VAX-VMS System. UNIX was chosen due to the rich set of development tools already available for it and the ease of developing new tools and applications. A VAX running the VMS operating system was chosen to demonstrate the handling of heterogeneous systems.

2.3.6. Cronus Clusters and the Internet

The goal of the Cronus project is development of a local area network-based distributed operating system. The Cronus cluster operates in the Internet environment as a class B network. Cronus hosts support the DoD Internet Protocol (IP) for datagram traffic, and, where connections are required, the DoD Transmission Control Protocol (TCP).

Cronus clusters is to use the Internet environment in two ways. First, access is provided to Cronus from points in the Internet external to the cluster. Second, the Internet supports communication between distinct Cronus clusters.

[2]One of the functions we would normally install on a GCE is the Cronus Internet Gateway, although it is currently installed on a DEC LSI-11 computer instead, because the standard Internet Gateway implementation uses the LSI-11.

2.3.7. The Advanced Development Model

The Advanced Development Model (ADM) of Cronus is the first instantiation of the Cronus hardware and software. It is, as its name suggests, the development testbed for Cronus. The ADM is experimental and changes as Cronus continues to be developed and as software is implemented, altered, and improved.

The ADM is being assembled using many off-the-shelf commercial hardware and software component building blocks. This reduces the cost of its components, permits the use of newly available state-of-the-art hardware, and enables us to be more flexible in its design. The design is flexible, to permit later substitution of more suitable hardware and software for deployable configurations.

3. System Overview

A distributed operating system manages the resources of a collection of connected computers and defines functions and interfaces available to application programs on system hosts. Cronus provides functions and interfaces similar to those found in any modern, interactive operating system (see the Cronus Functional Definition and System Concept Report [BBN 5879]). Cronus functions, however, are not limited in scope to a single host. Both the invocation of a function and its effects may cross host boundaries. The distributed functions which Cronus supports are:

- generalized object management
- global name management
- authentication and access control
- process and user session management
- interprocess communication
- a distributed file system
- input/output processing
- system access
- user interface
- system monitoring and control
- tools.

In this section, we introduce the Cronus design and briefly discuss the major elements of the system decomposition.

3.1. System Concept

The primary design goal for Cronus is to provide a uniformity and coherence to its system functions throughout the cluster. Host-independent, uniform access to data and services forms the cornerstone for resource sharing. The design of Cronus is based on an abstract object model. In this model, we treat the system as a collection of objects organized by type: files, processes, directories, and so forth. Only a limited number of well-defined operations can be invoked on an object, and the only information that a client can have about the structure or content of the object is obtained through these operations. The system structure is defined by the objects which constitute the system, the operations on these objects, and the responses which the objects give to the operations. The underlying structure of the system, which is essentially hidden from the clients, consists of the primitives which deliver the operations to active objects (processes), or to processes which are responsible for passive objects like files.

The Cronus distributed operating system is built from a number of concurrently existing objects called processes that reside on hosts which are part of the cluster. Some of them, called object managers, play a special role in implementing other objects of the system. Other processes provide services and specialized functions for the clients of the system. Still other processes run user programs. Processes communicate with each other to form larger abstractions and build more complex objects. At the most fundamental level, communication between processes is through messages sent over a local area network connecting the hosts of the cluster.

There are four interrelated parts to the Cronus system model:

- A *kernel* which supports the basic elements of the object model: processes, communication between objects, object addressing, and the relationship between objects and their manager processes. This part of the system includes facilities for locating an object and controlling access to it.
- A set of *basic object types*, along with the object managers which implement them. There are two groups of basic object types. One group is fundamental to the development of new object managers in Cronus. This group of object types includes: processes; principals, which identify system users; and symbolic name directories. Another group of basic objects is provided to support various application domains and processing requirements. Initially for Cronus this includes files and I/O devices.
- A *paradigm* for building and accessing new types of objects, which spells out the methods for integrating new object managers.
- *User interfaces* and related utility programs to provide convenient access for both people and programs to the system objects and services.

3.2. The Cronus Object Model

The object model provides a coherent and uniform framework for the system components of Cronus, and for application programs in a Cronus cluster. Since a distributed operating system is itself a distributed application, the methodology used in its construction should apply equally well to the construction of other distributed applications. The references [Xerox 1981, Rentsch 1982] discuss the object-oriented model of programming. The following are the key features of the object-oriented model that Cronus supports:

- Each Cronus object is a member of a well-defined class, which is called the *type* of the object. The names of Cronus types begin with the string 'CT_'; a list of some of the more important types may be found in Table 3.1.
- There is a set of *operations* (often called *methods* in the literature) defined for each Cronus type. These define the only ways that an object can be examined or modified.

- Every Cronus object has a unique identifier (UID) name. References to the object are generally through its UID, which is a bitstring uniquely identifying the object over the entire Cronus cluster. Cronus also has a symbolic catalog, mapping alphabetic names to UID's, to provide convenient reference to objects.
- The primitive *Invoke* causes a named operation to be performed on a named object.
- There is a basic set of operations (called *generic* operations) which are defined for all objects; these operations promote a unity among the various object types of the system and constitutes a limited form of inheritance of the operations defined for the basic type CT_Object. These operations include those which create and remove objects, and those which control access. Each Cronus type then has its own operations, and may redefine operations which are known to its parent class.
- An object has one or more parts that are visible to the outside world. These may include data, an object descriptor, and an active (or process) component. All Cronus objects have at least an object descriptor, which is the repository for such information as access rights.

Object Name	See Section
CT_Object	4.2
CT_Host	5.2
CT_Primal_Process	5.3
CT_Principal	7.5
CT_Group	7.5
CT_Authentication_Data	7.5
CT_Cronus_Catalog	8.2
CT_Catalog_Entry	8.2.2
CT_Directory	8.2.1
CT_Symbolic_Link	8.2.3
CT_External_Link	8.2.4
CT_COS_Directory	8.5
CT_Cronus_File	9.1
CT_Primal_File	9.2
CT_Reliable_File	9.3
CT_COS_File	9.4
CT_Line_Printer	10

Cronus Objects
Table 3.1

Fundamentally, the implementation of the Cronus system kernel consists of the implementation of the primitive *Invoke*. Each object is associated with an *object manager*, which knows all the internal details of the construction and location of the object. When an operation is invoked on an object, the Cronus kernel is responsible for delivering the operation to the appropriate object manager, which performs the task requested in the operation, and, if appropriate, responds to the invoker.

The operation switch in the Cronus kernel supports both invocations of operations on objects and message communication between processes. Since processes are system objects with defined operations to send and receive messages, the operation switch provides a host-independent interprocess communication (IPC) facility for both the system implementation and application programs. Further details of the object model and the design of the operation switch are described in Section 4.

Some of the attractiveness of a distributed architecture is the potential to exploit the redundancy and configuration flexibility of the hardware architecture. Cronus supports a unified approach to these attributes through its object orientation and by implementing a dynamic binding mechanism for routing operation requests to the appropriate object. In general, the location of the objects will be maintained in one of three ways. These are:

1. Primal Objects

   These objects are forever bound to the host that created them. There is no simpler form of Cronus object. An example would be a Primal File, which is permanently bound to its storage site.

2. Migratory Objects

   These objects may move from host to host as situations and configurations change. Standard Cronus mechanisms locate the current site to complete an object access.

3. Structured and Replicated Objects

   These objects have more internal structure than a single uniquely identified object. For example, a replicated file would have a number of primal files as its constituent parts. The UID would be recognized by manager processes on each of the sites for the more primitive elements. Replicated objects are a key element in Cronus system survivability, since availability to the objects continues as long as a sufficient subset of the copies are available.

Cronus can be extended by adding new object types to support new requirements or functions. Certain features are required for each object type including supporting the generic operations. In addition, for a new type that is similar to an existing type, many operations and their implementation may be inherited from the existing type, thus reducing the amount of work required to develop the new type.

The object model and its associated system components define a number of system conventions such as, integration with the monitoring and control software which may be adopted by subsystem designers, on a case-by-case basis. A subsystem designer can depend upon the existence of required features in other system components, and is obligated to provide them in each new component. The Cronus system design minimizes the number of required features for system entities, which, in turn, reduces the buy-in costs for

new hosts and object types.

Maintaining the integrity of complex objects is the responsibility of the managers for the type. This means that techniques can be tailored to the patterns of access to the object being maintained.

Since the generic operations include those which manage access permissions, uniform access control is a basic part of the Cronus object model. The object managers control access to the objects they maintain through the use of access control lists (ACL). The operation switch reliably stamps the UID of the invoking process on each of its messages, so the process making the request can be reliably identified.

The conventions for communication are based on the message structure library (MSL). A message consists of key-value pairs. There are also conventions that provide simple transaction protocols, and other features to support flexible message handling and processing. The MSL also standardizes the representation of data types, which allows the common interpretation of data items across a Cronus cluster. The MSL design is discussed in Section 6.

3.3. System Objects

To provide the initial operating capability, a number of basic system object types and their managers exist to support the functions outlined in the Cronus Functional Definition [BBN 5879]. They include:

- Process objects and process managers that support the Cronus system and user programmable processes. They may be linked together across the cluster, and connected through interprocess communication to form a user session.
- User identity objects and a permanent user data base that support authentication and access control.
- Directory objects and catalog managers that implement the global symbolic name space.
- File objects and file managers that provide a distributed filing system which can be used in providing non-volatile storage for developing portable object managers, as well as for satisfying application program data storage requirements.
- Device objects and device managers that support the integration of I/O devices into Cronus.

Much of the Cronus design has been decomposed into the subproblems of developing the Cronus distributed object model and of designing the components which provide these basic system objects.

3.4. Cronus Name Spaces and Catalogs

Cronus has two system-wide name spaces for referencing objects. The unique identifier (UID) for an object is the basic name. Unique identifiers are fixed-length, numeric quantities, intended for use by programs but unsuitable for people to read, remember, and type. The unique identifier has internal structure which Cronus uses, but is normally invisible to applications. It contains the name of object's type and the name of the host that generated it. The host name is useful as a hint for locating certain objects which do not migrate.

The Cronus system also includes a global symbolic name space oriented toward human use. Normally, the accessing agent would interact with the Cronus symbolic catalog manager to look up the unique identifier for the object. After it obtains the UID, the accessing agent can then invoke operations on the object.

3.4.1. Unique Identifiers

Although there is no single identifiable catalog supporting the UID name space, the notion of a catalog for UIDs is a useful abstraction. This catalog will be referred to as the *UID Table*; in practice, the functions that it supports are implemented by object managers for different object types by means of UID-to-object-descriptor tables, which can be thought of as fragments of the UID Table. When a Cronus object is assigned a UID a. :ntry is created in a UID table. This entry contains the information that the manager needs to access the object.

The Cronus operation switch provides client processes with addressing based on the UID, so if a client process has the UID, it can communicate with the object. The UID is a universal name that can be used from any one of the hosts in the cluster to refer to the object, no matter where in the cluster it is stored. Although it may not happen often in practice, objects may migrate from one host to another. When an object is relocated in this fashion, its UID does not change. A replicated object also has a single, unique identifier for client access to any of its images. Replicated objects may be developed out of more primitive, non-replicated objects which are usually accessed directly only by the replicated object manager

A Cronus unique identifier actually consists of a pair

< UNO, Type >

where *UNO* is an 80-bit unique number, and *Type* is a 16-bit value naming the type of the object. The UNO portion of the UID is uniquely associated with a particular object. All types are statically well-known and manually assigned, in the current system. This can be adapted to support dynamic types at a later time by using a portion of the 65,536 distinct types.

Each Cronus type has a generic name associated with it; this is a UID that has the type portion set to the type of the object and UNO portion set to zero. Cronus generic names are used for a variety of purposes. They act as class names in many of the places one would expect, particularly when an object is being created. That is, the creation of an instance of a class is treated as an operation on the generic name. In addition, the generic name is used when the system is interrogating the operation switch to find

a manager for the type. Generic objects are also used when the operation applies to an unidentified subset of objects, such as when all the objects of a particular type are searched to find ones with particular characteristics.

The operation switch is responsible for identifying the process that manages objects of a particular type. It does this by examining the type portion of the UID name on which the operation has been invoked. These managers are themselves Cronus process objects, which have UIDs of type CT_Primal_Process and UNOs selected when the process was created.

The facility that generates unique numbers may be regarded as existing continuously throughout the life of a Cronus configuration, and is accessible to system and application processes. No two requests by client processes for a UNO ever obtain the same UNO. Hence the unique number generator is an example of a survivable distributed program. The generator must be survivable, because UIDs must be unique over the lifetime of the cluster, and it must be distributed, because without it new objects cannot be created, so it cannot depend on any single host being up.

The UNO consists of three fields: a *HostNumber*, a *HostIncarnation* and a *SequenceNumber*. The HostNumber is the Internet address of the host that generated the UNO. The SequenceNumber is incremented for each request. The HostIncarnation is incremented if the SequenceNumber overflows its field. It is also incremented whenever a host is restarted. In order to assure that UNOs will never be repeated if a host crashes, the HostIncarnation is kept in stable storage, either on the host itself or on some other host that supports stable storage so the old value will not be lost.

The UNO size, 80 bits, was derived from assumptions about the number of UNOs that could be generated over the lifetime of the Cronus implementation and the mean rate at which systems enter or and leave a cluster. The current field sizes will allow a mean generation rate of about 10,000 UNOs per host per second and a mean crash rate of once every 3 minutes for 100 years; these numbers are assumed to be adequate for reasonable system activities.

3.4.2. Symbolic Names

The principal design consideration for the symbolic name space is to make it easy for people to use. Names for Cronus objects are uniform and host independent. Symbolic names are supported by a catalog that provides a mapping between symbolic names and the UIDs. This name space is a tree, composed of nodes and directed labeled arcs. The base is a node called the *root*. A complete symbolic name begins with the punctuation mark colon (:), representing the root node, followed by the names of the arcs, separated by colons. For example, :a:b:c is the symbolic name of an object. Nodes in the tree generally represent Cronus objects which have symbolic names, such as files and catalogs. Nodes may also be symbolic links to other catalog entries.

Not all Cronus objects have symbolic names, and those that do may have more than one. When an object is given a symbolic name, an entry is made in the Cronus Catalog, and when the name for an object is removed, its entry is removed from the Cronus Catalog. The Cronus Catalog supports Enter, Lookup, and Remove operations. In addition, operations are provided to read and to modify the contents of catalog entries.

The catalog is distributed; different hosts manage different parts of the name space. The implementation is logically integrated, however, so that any catalog manager process can be asked to perform any of the catalog operations. Portion of the hierarchy may be selectively replicated to support efficient or reliable access to different parts of the name space. The Cronus catalog is described in detail in Section 8.

3.5. The Cronus File System

The collection of all Cronus files constitutes the Cronus distributed file system. Within this file system, Cronus supports several file types. The most basic file is a primal file, which is stored entirely within a single host and is bound to that host throughout its lifetime. Other types of Cronus files are built from primal files. A replicated (or multi-copy) file, which has multiple instances replicated across Cronus hosts for increased availability or enhanced responsiveness, is constructed from several primal files. Therefore, if a host contributes storage resources to Cronus, it must support primal files.

There is no single table that lists all file objects. Rather, each file manager owns all of the data for the file objects it manages. The Cronus object addressing facilities make possible a client interface in which knowledge of a UID is sufficient to access the file regardless of its location. Clients may make file placement decisions themselves if they wish. Otherwise, file placement is chosen automatically after evaluating available files and file manager resources.

Ordinary read and write operations may be performed on file objects. The expected mode of access to Cronus files is to transfer the file data as needed, much like conventional filesystem access to disk files. Copies of Cronus files are made only to satisfy explicit user requests or to support other system requirements. The design for the Cronus File System can be found in Section 9.

3.6. Cronus Process Management

Primal processes are the simplest process entities. They are constructed from the process abstraction that exists in the constituent host operating system. This simple form of process is used as a building block for the system implementation, minimizing integration costs for new Cronus host types. Since primal processes cannot be loaded dynamically with user programs and lack flexible process control functions, they are too inflexible to be used as vehicles for general application programming, but are used as object managers and in other well-defined system roles.

Cronus processes have most of the features natural to the host on which they are built, and no attempt is made to hide these features. An application builder has the choice of when to use locally-supported features and when to use standardized Cronus features. To the extent that applications choose to adopt Cronus process features, they will be better integrated with the other cluster processing activities. On the other hand, the judicious use of local features will enhance the efficiency of the activity. Cronus processes are described in Section 5.

3.7. Device Integration

Special purpose devices, such as line printers and tape drive devices are important elements in a system configuration. As Cronus objects, these devices are available to the entire cluster through an object manager. In some cases, more elaborate interfaces can provide an access path with specialized features. For example, a line printer service, can be provided that supports spooling. Device integration is discussed in Section 10.

3.8. User Identities and Access Control

Users are represented by system objects, known as *principals*. A principal object contains data that describes the manner in which the user may use the system. This information supports operations such as authentication and session initialization. The object manager for the principal objects and for other access-related objects is called the Authentication Manager. The Authentication Manager component services the entire cluster.

The purpose of Cronus access control is to prevent unauthorized access to Cronus objects. This is done uniformly by associating an access control list (ACL) with each object. Access is then either granted or denied based on the identity of the principal associated with the accessing agent and the contents of the access control list for the object.

The operations of the Authorization Manager and the access control system are discussed in Section 7.

3.9. Process Support Library

The Process Support Library (PSL) is a collection of functions, that may be bound into the load image of a Cronus process.

PSL routines are considered part of the Cronus system and are generally supplied with the system and maintained by system programmers. Thge PSL fills the following major roles:

1. It provides a convenient interface to Cronus operations.
2. It provides access to special Cronus features such as the facilities which generate UNOs and structure messages, and to the elementary file system that underlies the primal file system. It also provides a uniform interface to the interprocess communication facility. These features are not normally accessed though the Operation Switch.
3. It provides COS interface and utility routines necessary to support the production of portable programs. This includes format conversion routines and defines machine-dependent constants.

3.10. Important Subsystems

Subsystems are components which use system-provided features to support user services. Two important subsystems in the initial implementation of the Cronus systems are the user interface and the monitoring and control subsystem.

The user may gain access to the system from dedicated terminal access concentrators, from one of the shared hosts, or over the internet. The interactive processes which are controlled by the user interface will be distributed across the cluster as required either by the application itself or under the direction of the user. A discussion of the user interface may be found in Section 11.

The monitoring and control subsystem (MCS) makes it possible for an operator to monitor and control the entire cluster configuration from a single console. The functions of the MCS include starting or restarting parts of the Cronus configuration, monitoring its facilities and components, and collecting error reports and statistics. The MCS monitors object managers and collects statistics based on a functional decomposition across the Cronus configuration rather than a site-based decomposition. The monitoring and control design is described in Section 12.

3.11. The Layering of Protocols in Cronus

The underlying support for the Cronus cluster architecture is a local area network. The Ethernet standard has been selected for an inter-host transport medium within the initial Cronus configuration. The Cronus design does not, however, depend directly on this, so later versions may use a different local network. Furthermore, the design does use the DoD standard protocols at higher levels, and requires an interface between them and the physical local network.

To accomplish these objectives, we have developed a Virtual Local Network based on DoD Internet Protocol (IP) conventions and a representative set of local area network capabilities. The Virtual Local network is an interhost message transport medium which is independent of the physical local network.

The Virtual Local Network layer is described in section 14. It provides a primitive datagram service, compatibility with Internet addressing, and independence from the details of the physical local network. VLN datagrams can be specifically addressed, broadcast, or multicast.

4. Object Management

4.1. Introduction

In this section, we outline the Cronus object model and show how it is used to structure the kernel of the system. This discussion consists of the following elements:

- A short discussion of the object model in general, and of its relationship to Cronus objects.
- A general description of the basic objects that are included in the first implementations of Cronus.
- The system primitives that Cronus uses to cause operations to take place on objects.
- The role of special processes, called *object managers*, in the implementation of objects.
- The mechanization of the Cronus primitives, and the role of the *operation switch* in this mechanization.
- The definition of *generic* operations that are defined for all Cronus objects.
- The structure of object managers.

In the course of this section, it will be necessary to refer to the characteristics of Cronus processes, and to the methods of communicating between such processes. Those elements of process management and interprocess communication which are needed for the understanding of the Cronus object model and for the construction of object managers will be sketched in this section, while the details have been placed in Sections 5 and 6.

4.2. General Object Model

There is a considerable and growing literature concerning object models and object-oriented programming, and it is not our purpose to describe these methods in detail. On the other hand, the conceptual framework and terminology of object-oriented programming and system decomposition has not fully stablized, and any system, like Cronus, that claims to use this methodology is actually selecting from a range of ideas and applying them to a specific situation; in this case, to the design and implementation of a distributed operating system.

The basic idea of object-oriented systems is that all interactions can, at some level, be described in terms of a set of defined operations on objects. These methods are strongly associated with the development of the Smalltalk-80 system [Goldberg 1983], but are also an outgrowth of work in the manipulation of data abstractions [Liskov 1977], [Robinson 1977], and recent developments in programming languages. There are useful, brief introductions to the use of these methods in [Jones 1978], [Weinreb 1981] and [Rentch 1982].

At first glance, one might consider it enough to think of an object as an instance of a data abstraction. If the internal structure of the data object is suitably hidden from the outside world and the proper operations provided to manipulate the object, we can find out everything we need to know about it and, equally important, nothing about how the object is actually put together. This is a strong application of the hiding principle of software engineering, combined with a set of methods to examine and modify the part of the data object which is of interest to the outside world.

The object model is this and more, however. There are several extensions to this basic idea which have been made in various systems. One of the most important is *inheritance*, which we will discuss below. Another is the addition of objects which are more than instances of a data abstraction; for example, in Cronus we have process objects as well as pure data objects.

In Cronus, all the objects which are alike in their structure and in the operations which they respond to are members of a Cronus *type* (in other systems, this is often referred to as a *class*). Inheritance describes a relationship between types. We can say that a particular type is a *subtype* S of some other type T. In saying this, we are saying that an instance of the type S is like an instance of type T in some important way. Usually this is described by noting that any operation which may be invoked on an instance of T may also be invoked on an instance of S. This does not mean that exactly the same procedure will be applied to exactly the same kind of entity. For example, all Cronus objects inherit the properties of the basic Cronus object type CT_Object. There are a set of operations defined on this object, including Remove, which causes the object to go away. A very different procedure is used to Remove a primal file object than the one which removes a user process. But there is some clear intuitive feeling which we have of what Remove means if we think of primal files and user processes as objects.

It is worth noting that the inheritance relationship is rather different from the relationship which one finds in composite objects. For example, the Authentication Manager supports the type CT_Group, which is a composite object that is built out of principals (objects of type CT_Principal, which is a representation of a system user) and other objects of type CT_Group. Groups are not subtypes of principals, but are constructed from them. Some operations that can be invoked on a principal, such as the ones which manipulate the group expansion list have no analogue in the definition of a group, and make no sense if they are invoked on a group.

The following are the basic object types that constitute the initial implementation of Cronus:

CT_Object: This is the most basic type, and the generic operations that create and remove objects and maintain the access control lists and object descriptors are defined for objects of this type. In Cronus this is an entirely abstract form, and there are no instances of objects of type CT_Object.

CT_Host: The Cronus system is made up of a series of hosts which provide services for users. This object has a process component that creates and manages the primal processes that, in turn, actually perform the services and manage the other objects of the system. The CT_Host object is sometimes called the Primal Process Manager for the host, because that is its most visible function. The CT_Host object is closely allied with the operation switch, which is used to implement the invocation of operations on objects.

CT_Primal_File: The initial implementation of Cronus supports files which are bound to a specific host. All ordinary user data is stored in objects of type CT_Primal_File. In

addition, a number of other object types are constructed from primal files.

CT_Directory: The Cronus catalog is formed from a tree of objects of type CT_Directory. The internal structure of each directory is entirely hidden from the user by the Catalog Manager.

CT_Principal: A principal is the system's representation of a user or a system service which requires access to some other service or object manager. The access control system depends on identifying the objects of type CT_Principal which are permitted to carry out an activity.

There are a number of other object types which are associated with the Catalog Manager (such as CT_Symbolic_Link) and with the Authentication Manager (such as CT_Group), but the system could function without them.

In object-oriented programming, a client invokes operations on an object, often called the *receiver*, which is identified by a UID, *ObjectUID*[3]. The operation itself may be represented as a pair

<OperationName, Parameters>

In Cronus the basic primitive which causes an operation to be invoked on an object is *Invoke*. This causes Operation to take place on the object named by UID. The operation switch of the Cronus kernel provides for delivering the request to a manager for the named object (see Section 4.5).

While the primitive *Invoke* is sufficient to support the system, the relatively large number of reply messages suggest that there should be a more efficient method for answering a request[4]. A second message primitive, *Send* is provided for this purpose. When a message from a client is delivered, the process UID for the client is included. The manager may then use *Send* to reply directly to the client.

In a distributed system, the client does not usually know which host has the object manager which is responsible for a particular object. To allow objects to be dynamically located, there is a particular operation, called *Locate* that is among the operations defined for every object in Cronus. When this operation is invoked on the object UID at a particular host Address, the object manager for that type will reply if it manages that object[5].

If the client does not specify the host when invoking the operation, the Cronus kernel performs the required Locate operations to determine where to send the operation. These Locate operations are often performed using the broadcast facilities of the VLN. The kernel or the client may cache locations of specific objects and object managers for increased efficiency. In addition, primal objects, which are bound to the host which creates them, can be found quite easily host address portion of the UID contains the address of the host which generated the UNO portion of the UID. For the current implementation, the

[3]There are a few cases in Cronus where objects are identified by other means, for example, a specific catalog entry may be identified by the symbolic name which is being manipulated. The argument presented is analogous, so it is sufficient to consider the cases where the object actually has a UID.

[4]If Invoke is all that is available, the reply must be passed through the process manager for the process to which the reply is directed. Send allows the reply to be routed directly to the client by the Cronus operation switch.

[5]Actually, if the client wants the negative acknowledgement, it will also reply if it doesn't manage the object.

UNO is generated on the host that creates the object, and that also currently holds the object if it still exists.

Subtype relationships are not a primitive concept in the implementation of Cronus. There is no direct implementation of inheritance; there is, instead, a discipline which says that the manager of each subtype must implement the inherited operations. In addition to simplify implementation of the inherited operations (which is used for the generic operations), there are several static implementation techniques that achieve inheritance. A manager may register several type values with the operation switch, and implement some as subtypes of the others internally. Alternatively, one manager may invoke another through the standard mechanisms.

4.3. Object Naming

The Cronus object model requires a mechanism for delivering messages addressed to objects. This mechanism, outlined briefly in Section 4.2 and described in detail in Section 4.5, is called the operation switch. The operation switch, in turn, requires the client to identify the object which is being modified or examined. The standard identifier for an object is its UID, which is a bit-string containing 96 bits. This bit string consists of two components: a unique number (UNO) that is different for each object which has ever existed in the cluster, and the Cronus type. It is useful to think of the UID as having four fields:

HostAddress: the 32-bit Internet address of the host which created the object. If the object is a primal object, the HostAddress is also the actual address of the object, if it still exists.

IncarnationNumber: a field containing an integer which is incremented whenever the host is loaded or reset, or when the associated SequenceNumber field overflows.

SequenceNumber: a simple counter field which is used to assure the uniqueness of each UNO that is used to name an object.

CronusType: the 16-bit integer specifying the Cronus type of the object.

Between them, the IncarnationNumber and SequenceNumber fields contain 48 bits, but the subdivision of this string may vary from host to host; for the hosts in the initial implementation, each field is 24 bits long.

It should be observed that the object is actually identified uniquely by the UNO portion of the UID, and that the Cronus type is added so the operation switch can find the object manager. In particular, it is possible to think of an object as having more than one UID, consisting of the same UNO paired with different types. The current system does not make any interesting use of this possibility.

There are also generic (or logical) names, which consist of a zero UNO and a type field specifying the type of the generic name. Specific names are used for objects which can be created and destroyed, and have private state information which is important to the accessor (e.g., a particular file). Generic names are used for special purposes. For example, the client can find out if there is an object manager for a particular type on a host by invoking *Locate* on generic name. Generic names are also used in operations, like Create, in which there is no object name available; the generic names act like *class* objects

in other object oriented systems like Smalltalk, or like the generic addressing facility in NSW's MSG, which is used to address an instance of a service.

Operations applied to generic names may specify a particular host. The *ReportStatus* command can be invoked in this way to request the status of the manager of the given type on the specified host. The *Create* command, used this way, would create an instance of the given type on the specified host. When the host is not specified, the managers may consult with each other and use resource management policy parameters to determine where the operation should be performed or where a new object instance should be placed.

Accessing agents interact with object managers using Cronus Interprocess Communication. The client may initiate access by giving either the UID for the object or by giving its symbolic name. The PSL provides functions which will accept either name. If the accessing process has the UID of the object, the PSL simply constructs a message that invokes an operation upon it. The operation switch delivers the requested operation code, the UID, and any other parameters to the appropriate object manager. The object manager consults its fragment of the UID Table to access the object as necessary to perform the requested operation. If, on the other hand, the accessing process does not have the UID, the PSL first consults the Cronus catalog; then, when it knows the associated UID, it composes the message and sends it on its way.

This means that we allow the symbolic catalog to be by-passed when an object is accessed, and the accessing process knows the UID. This improves performance and enhances the flexibility of using primitive objects to build complex objects, since the object manager for the complex object can use the UIDs of its components directly. The cost of achieving these benefits is primarily one of increased implementation complexity:

1. Access control is performed in a decentralized fashion by all of the object managers.
2. Information about objects is distributed among object managers and catalog managers. Care, must be taken to ensure that the information about an object is consistent, or if it is not, that the system can operate properly.

4.4. Generic Operations On Objects

The generic operations are defined for all system objects. These operations fall into several groups:

Create and Remove: These bring an object into existence and destroy it. The operation Create is invoked on the generic name for the object. These operations must be defined for all objects

Locate: If the object exists and is managed by the object manager which receives the message, the manager replies that it knows about the object. This operation must be defined for all objects.

ReadACL and WriteACL: These manipulate the access control list of the object. These

operations must be defined for all objects which are separately access controlled. There are a few objects whose access is controlled through another object. For example, objects of type CT_Catalog_Entry are controlled through the permissions on the containing object of type CT_Directory.

ReadSysParms, WriteSysParms, ReadUserParms, WriteUserParms: Every object has an associated object descriptor. The object descriptor contains various pieces of information about the object that are made visible to the outside through these Read operations, and may be modified by the Write operations. Access is controlled separately for the User and Sys portions of the object descriptor.

ReportStatus: This operation is normally performed on a generic name associated with an object type. For example, ReportStatus is invoked on the generic CT_Primal_File to find out how much space there is available on the associated file system.

For some operations, such as Create, the exact list of parameters and responses will vary from object type to object type. Other operations, such as those which operate on the access control list, perform in the same way for all object types. For details, see the appropriate sections of the Cronus User's manual, especially object(3), acl(3), the descriptions of the objects below and in Section 3 of the Cronus User's manual, and the descriptions of the PSL routines in Section 2 of the Cronus User's Manual.

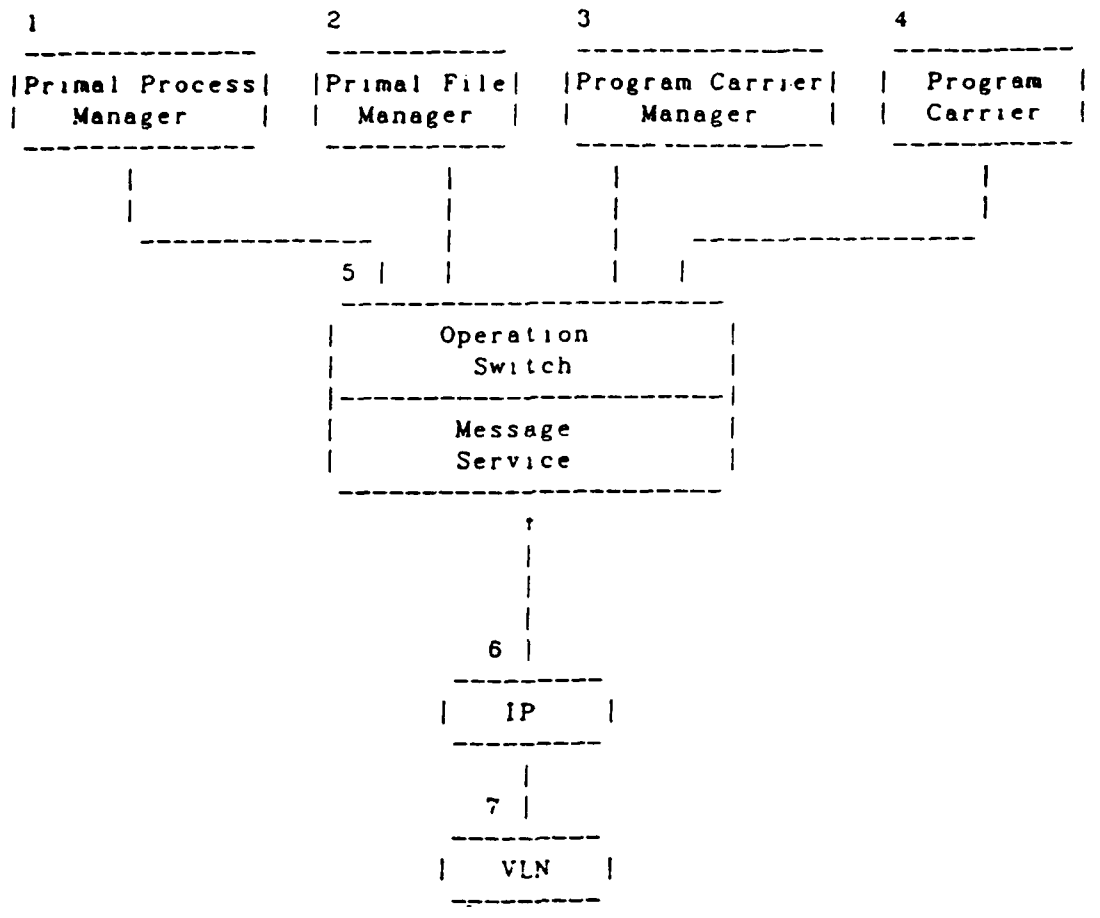
4.5. Object System Implementation

In order to describe the design of the operation switch and its role in message-oriented interprocess communication, we must briefly introduce Cronus processes (the Cronus process is described in detail in Section 5).

Cronus processes are constructed from *constituent host processes* (CHPs). The properties of a CHP are defined by the machine architecture and the constituent host operating system (COS). The Cronus process is constructed from one or more CHPs, with the addition of Cronus process features. The simplest type of Cronus process is the *primal process* (PP). A primal process is a CHP which can invoke operations on objects through the Cronus Interprocess Communication facility and can be controlled by the Primal Process Manager. In addition, a primal process can use the Cronus primitive Receive to receive messages sent through the Cronus IPC by either *Invoke* or *Send*.

The implementation of Receive employs CHP-specific synchronization facilities to build an asynchronous Receive operation.

This section describes the framework of the object system implementation on Cronus hosts. Figure 4.1 illustrates the relevant components on a single host. The boxes in the figure represent abstract modules of the implementation, and do not necessarily map one-to-one into CHPs or address spaces.



Object System Components
Figure 4.1

In Figure 4.1, boxes 1-4 are Cronus process objects: box 5 is the operation switch, which accepts messages from and delivers messages to the Cronus processes on this host; box 6 is the IP protocol demultiplexing service; and box 7 is the Virtual Local Network layer.

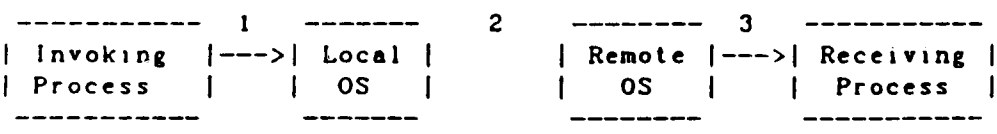
The operation switch is table-driven. This table contains routing information that the operation switch uses to direct messages from process to process. The sender and receiver may both be on a single host, or the message service may be involved in a host-to-host message transfer. The operation switch does not retain information about the messages, although it may gather statistics and transmit them to the Monitoring and Control System (see Section 12).

Since the invoker can request reliable message transport, and ordinarily does so for InvokeOnHost applied to a specific host address, a failure of an operation invocation is not likely to be due to a transient communication fault, with high probability, either the network or the target host, or both, are down (see Section 6 for a detailed description of the IPC and these services).

The invocation sequence for an operation is:

- The Cronus Process Support Library (PSL), which is the component of the system that appears within the client process, formats a message which contains the name of the object, the operation, its parameters, and other information which is needed by the system.
- The message, which is marked as an invocation of the operation, is handed to the local host's operation switch. If HostAddress specifies the local host, it processes the message itself; otherwise, it forwards the message to the specified host. When no specific host is indicated the operation switch will issue a Locate to find a manager for the specified type and route the request to one of the managers that reply. (These functions are directly supported by the Cronus Interprocess Communication facility, which is described in detail in Section 6.)
- The receiving operation switch examines the ObjectUID, determines the type of the object, and hands it to the object manager for that type, if there is one. If the receiving manager supports resource management, it may consult with other managers, and choose the one best suited to perform the request. If the manager itself is best suited to handle the message it will do so without any additional transactions. Otherwise it will forward the request to the selected manager, indicating that the selected manager should perform the request without any additional consultation.
- The object manager for the object type then performs the operation indicated by the operation and its parameters.
- Although it is not necessary for an operation to follow a request-reply paradigm, most do. If a reply is needed, the object manager prepares a message that is returned using the *Send* primitive.

Figure 2 illustrates the transmission of an operation from the invoking process, through the local operation switch, to the remote operation switch, and finally to the receiving process. This section



Operation Switch Interfaces
Figure 4.2

describes the calls and the representation of data structures at the interfaces 1, 2, and 3.

When the client performs an *Invoke* primitive on the Cronus object, a message is generated that is ultimately directed to a manager process and accepted by a Receive in that process. Information crosses interfaces (1) and (3) by means of Cronus system calls, which are representations of the primitive functions, made by the invoking and receiving processes; these calls may be represented as:

Invoke(TargetAddress,ObjectUID,Operation)

Receive(SourceAddress,SenderUID,ObjectUID,Operation)

where the function parameter Operation includes both the intended operation and its parameters[6].

Interface (2) is peer-to-peer communication between operation switches, which is discussed in greater detail in Section 6. Messages exchanged between operation switches are octet sequences. The Operation parameter of the Invoke call is not interpreted by the operation switch, and is treated simply as data to be moved. The message has several header fields that are visible to both operation switches; these include the UID of the object being operated upon (ObjectUID) and of the client (ProcessUID).

When the Invoke message arrives at the target host, the operation switch tries to map the type to a manager process on the host. The table of possible destinations consists of a list of generic UIDs for ordinary managers and specific UIDs for objects which are managed separately [7]. The operation switch first checks the ObjectUID against the list of specific UIDs, then the Type field against the list of generic UIDs. If the mapping is not successful, the invocation is discarded, but will generate an exception reply. If the mapping is successful, the message is transmitted to the manager process. The manager obtains the information by initiating an ordinary Receive request; when the Receive completes, the SourceAddress, InvokerUID, ObjectUID and Operation have been made available to the manager process.

Although one can reply by invoking the Send operation on the object ProcessUID, replies are usually sent by means of the alternative *Send* primitive. This primitive hands messages addressed to a specific process across interface (1). The operation switch then marks the message which it ships across interface (2) as a *Send* message. The receiving operation switch then places the message on the queue for the target process, bypassing its object manager. The mechanism for delivery, Receive, is independent of the transmission mode of the original message.

[6]The calling sequences for these functions have been modified for purposes of presentation clarity; see the Cronus User's Manual send(2) and receive(2) for a description of the actual calling sequence.

[7]Currently, the only example of such a separately managed object is the virtual terminal in the user interface (see Section 11).

4.6. Object Manager Structure

Object managers are asynchronous independent processes. They are asynchronous because they interleave the processing of messages. An object manager often invokes operations on other objects to satisfy the requests it receives; it does not wait for the reply to such a request, but moves on to the next request or reply from a previous operation. They are independent processes because they are daemon processes which are started by the system (or its monitoring and control section) or by another daemon process. They receive messages, originate requests to satisfy the client requests, and reply to the original messages.

The asynchronous character of the object manager has a significant impact on its structure. Managers receive messages which cause them to undertake actions. These actions may be of two types. The first type occurs entirely within the manager's own address space (or within a single Cronus process that may consist of more than one COS process), and is called a local action. The second type requires the manager to perform one or more operations, called secondary requests, on objects that it does not manage. It must be able to keep track of a number of these actions. On the other hand, the manager cannot wait for the response from a secondary request before it accepts its own next request. The processing that comprises the operation is divided into portions that are performed before and after the secondary request is issued. When the manager issues the secondary request, it saves components of its state that are needed to complete the processing when the reply arrives.

There are a number of common elements in the construction of object managers. Cronus manager development tools assist in the development of managers by producing code for these parts of the manager. The developer provides a simple specification of the type and its operations, from which the code is automatically generated.

> A manager normally consists of an initialization section and a main loop which is driven by the arrival of requests through the Cronus interprocess communication facility. Since a manager normally runs forever (until the system crashes), there may not be code for wrap-up.
>
> The manager parses incoming messages, and dispatches on the message class, which takes on the values *Request*, *Reply*, and *InProgress*.
>
> A new Request message causes the manager to set up a control block for the operation.
>
> A Reply message causes the manager to identify the control block associated with the message, and to continue processing as required by that message.

In the case of a local action, the manager receiving the message will (normally) process the request to completion and compose a reply to the originating process.

If a secondary request is necessary, the situation is similar to that found at the originator. A request can be put into the form:

```
InitialPortion
Op(Obj) -> Reply
PostProcessing
```

That is, a secondary request is basically some operation (Op) on an object (Obj) which generates a Reply. Before we invoke this operation, we usually have some initialization beyond composing the message (InitialPortion) and after we get the reply, we often need to do some PostProcessing.

The procedure that invokes the operation also creates a control block that contains the information required for reply processing. After it passes the invocation to the IPC mechanism, it returns without waiting. The manager then processes the next IPC message (which may be a Reply from a secondary request, or a new Request), if there is one available. Otherwise, it goes to sleep until the next message arrives (see Section 6). When a Reply for a secondary request arrives, the manager finds the control block associated with it, and performs the reply function. When the reply processing returns normally, the PostProcessing routine is invoked if the message is marked OK, and an alternate error-handling routine is invoked if the message is marked NOT_OK.

The independent character of the object manager principally effects the way errors are handled. When a process is interactive, it makes some sense to report the error to the user. If an independent process detects an error condition, it may be necessary to report the error to the client that issued the request, to the monitoring and control station (MCS, see Section 12), or to both. In addition, Cronus managers keep statistics on the kinds of errors which have been detected, and report them to the MCS periodically.

A manager that encounters a failure during an operation, particularly when there are secondary operations involved, must take steps to assure that the information which is retained across host crashes (the permanent state of the system) and any internal status information (the temporary state of the system) are correct and consistent.

5. Process Management

5.1. Introduction

Processes are the active portion of any system. Each host and constituent operating system in a Cronus cluster has at least one natural concept of the *process*. More generally, several different kinds of processes are present in each host, fulfilling different roles. In the absence of a distributed operating system, the processes on two hosts are unrelated to each other. This section describes how Cronus processes work and how they communicate with each other. In the following discussion, it is usually safe to visualize a Cronus process as being built from a single Constituent Host Process (**CHP**) with the addition of an object descriptor and some specialized facilities which make Cronus work. On the other hand, the implementation might be quite different in reality. That is, a Cronus process might be made up of several CHPs, or a CHP might include more than one Cronus process [8].

If we wish to build a system of cooperating processes on a cluster of computers, and to use it as a base for a distributed operating system, we must do the following:

- Define a standard method for communicating among the processes. Cronus treats processes as objects, and uses the standard Cronus IPC facility and the primitives *Invoke* and *Send* for all interprocess communication. All procedures developed for structuring and parsing messages for operations on objects, such as those described in Section 6, may be used for manipulating process objects as well.
- Establish mechanisms for creating and controlling processes on hosts of different sorts. Again, since Cronus processes are objects, this reduces to the definition of the operations which may validly be applied to the process objects.
- Provide a method for organizing the process objects to perform tasks. This is accomplished by defining other objects which reflect the required organization. The collection of processes on a host, for example, is represented by an object of type CT_Host, which will be described below.

The following Cronus types are discussed in this section:

- CT_Host: the organizing object for the primal processes associated with a physical host.
- CT_Primal_Process: the most fundamental type of process. Object managers are normally constructed from processes of this type.

There is one object of type CT_Host associated with each physical host, and it is the object manager of the processes of type CT_Primal_Process on that host. It is responsible for starting up Cronus services, which are also object managers for the basic system objects; it is also responsible for gathering the

[8] In fact, a Cronus process might even span hosts. In the current system design, all Cronus process are primal processes; that is, they are bound to a single host. Later implementations may relax this restriction.

information which the operation switch needs to route messages to the other object managers and to specific processes when the primitive SendToProcess is used.

Primal processes never migrate; once created, the process remains on the same host until it is destroyed. The HostAddress in a UID for a primal process tells where the process is, so an operation switch can tell exactly where to deliver a message addressed to it.

Every host participating in the system must support an object of type CT_Host, which is also referred to as a Primal Process Manager (PPM), and primal processes. In their minimal forms, the host object and primal processes are relatively simple. This keeps the cost of integrating a host type into a Cronus cluster low for those minimally integrated hosts that can obtain system services from other hosts, but do not provide system services.

A collection of primal processes which play a well-defined functional role within the system are collectiviely called a Cronus *service*. For example, the Primal File managers form the Primal File Service; the Primal, COS and other subtypes of CT_File, form the Cronus File Service.

Cronus processes may make use of some or all of the functions in the *Process Support Library* (PSL), which provides high level interfaces to many system functions as well as general purpose utilities for interfacing to and manipulating the Cronus environment. Portability is a major goal for the PSL, so that it can be implemented readily in whole or in part on new host types. The PSL is discussed further in Section 5.4.

5.2. Objects of Type Host

The basic organizational elements of Cronus are objects of type CT_Host. These objects correspond to the intuitive physical hosts that make up the Cronus cluster. A CT_Host object consists of the the Primal Process Manager for the host and the basic tables which are used by the operation switch in routing operation invocations. In some sense, it is reasonable to think of the operation switch itself as a part of CT_Host. When a host joins the Cronus network, only the lowest level of network software is functioning: the Monitoring and Control System (See Section 12) engages in a dialogue with this primitive host element, and brings up the object CT_Host. The MCS is therefore the object manager for the objects of type CT_Host.

The Primal Process Manager (PPM) component of a CT_Host object implements operations concerning primal processes as a class. The tables that identify the object managers and processes that are on a particular host, and that therefore are used to implement the Cronus primitives *Invoke* and *Send*, are maintained by the Register and Delete operations on the CT_Host object.

In addition to the generic operations, the following operations are defined on objects of type CT_Host:

CronusRestart
ListService
ListProcess
Register

Delete

The Cronus Restart operation is used to terminate all activity on the CT_Host object. It removes all active processes, including the process implementing the CT_Host object itself. After a Cronus Restart, the host is in a state from which it may be bootstrapped.

The ListService operation is used to find out what kinds of service the host is prepared to support, and which ones are in fact being supported. The names of these services, which are called role designators, are used to start primal processes that perform the service (see Section 5.3).

The ListProcess operation tells what processes are active and what roles they are playing; this is the information which the operation switch has about processes active on this host. Whenever a process is created or removed, the tables must be updated. These tables contain the following entries:

- generic names for objects paired with the specific UID of the Cronus process;
- specific UIDs for process objects that will receive messages through *Send*; and
- specific UIDs for those objects whose manager cannot be identified by reference to a generic name (see Section 11).

The tables also contain any COS specific information needed to communicate with the process. They are automatically updated for processes which are created by the CT_Host object itself, such as the object managers. Processes created by other managers inform the CT_Host of changes through the Register and Delete operations.

5.3. The Operations on Objects of Type Primal Process

Objects of type CT_Primal_Process are among the most basic in Cronus. The three system primitives (Invoke, Send, and Receive) are defined for these objects. In addition, the generic operations are defined. The particular characteristics of these operations, when invoked on primal process objects, are described in detail in the Cronus User's Manual.

The Create operation takes a role designator as an argument, and starts a new primal process performing this role. The role designator may be in one of the following forms:

1. A Cronus generic UID name for the service.
2. A Cronus symbolic service name. These are character strings containing the literal characters of a logical name, for example "PrimalFile".
3. A host dependent role designator. These are arbitrary strings, which have meaning only to the PPM on a specific host.

Role designators of kinds (1) and (2) are paired, and are registered with the Cronus system administrator as the names of standard Cronus functional units. The allowable list of role designators of these kinds for

a particular host object may be obtained by invoking the operation ListService on the object. These primal processes are automatically registered, which makes the logical name known to the operation switch on the host, so that the process can be generically addressed.

Designators of kind (3) provide for the activation of host-specific programs or devices. The host dependent role designator might be a COS-dependent file that is executed as a result of the Create operation. Primal processes created with a host-dependent role designator generally have no associated logical name, and cannot be generically addressed.

The primal process will initialize its state entirely from non-volatile storage (local or remote disks).

A process may invoke any operations on itself as the target object. A process may send itself messages, remove itself, or read or change its descriptor in the same way it performs these operations on other objects.

The operations defined on primal processes provide process control functions. For example, Remove is invoked to "destroy" or "kill" the process. It erases all record of the process state from the system and frees any resources dedicated to the process.

A process which is removed is not notified of the operation, and has no opportunity to terminate cleanly. Only the resources actually used to implement the process object are freed; resources held as a result of the computational activity of the process (e.g., locks on remote files) are not freed. Some primal processes may possess dedicated resources, and Remove disables the process, without releasing these resources.

A reply will be generated to the invoker to indicate that the process has been removed. After receiving the reply, the invoker knows that operations using the UID of the process will not succeed.

The *process descriptor* is the object descriptor portion of the Cronus process. It is useful to think of the process descriptor as a list of (key, value) pairs, in the sense of the MSL (See Section 6.2). Some of the values implement process control. For example, the pair (Key_Priority,5) would indicate the importance of a process relative to other processes for competing resources. Some keys must be present in the list ("required keys"), while others are optional.

All process objects must respond to the *required keys* in a uniform way. If an object supports a standard *optional key*, the process must apply it in a uniform, system-wide manner. Additional, *elective keys* may be present. Their interpretation is not specified by Cronus, but is the responsibility of the process and the other processes with which it interacts.

Currently, the required keys for Primal Processes are Key_MyUID, Key_MyAGS, and Key_IPCEnabled.

The value associated with Key_MyUID is placed in the descriptor when the process is created, and is never changed thereafter. It is the specific UID of the process, and has type CT_Primal_Process (or CT_Program_Carrier, in the case of program carrier objects).

The value of Key_MyAGS is the access group set, used with access control lists to determine access rights to objects at operation invocation time. The initialization and use of access control and authentication data is discussed in detail in section 7.

The value of Key_IPCEnabled controls communication through the operation switch. If the value is true, the process can send and receive messages in the normal fashion. If it is false, the process may not send or receive messages, or invoke operations on Cronus objects. This feature can be used for managing access to network resources.

Currently, the only optional key defined for a Primal Process is Key_Priority, but others may be defined later.

The generic operations on object descriptors permit a process to inspect or modify the descriptor of another process. If several processes invoke these operations on another process at the same time, the effect will be as if the operations were processed sequentially, i.e., they are atomic with respect to each other.

Since the CT_Host object is implemented by a Primal Process, these process control operations apply to it. One of the operations, Remove, has a special meaning when applied to the CT_Host. Because it is the manager of Primal Processes, removing the CT_Host removes all Cronus processes on the host. This forces a shutdown of the Cronus system on the host.

5.4. Process Support Library

The Process Support Library (PSL) is a basic part of the Cronus implementation. It contains a large number of functions which can be used to construct Cronus object managers and user programs. All Cronus programs are expected to use the PSL to perform the functions which it supports. The distribution of responsibilities between the PSL and the Cronus kernel is often not defined, and may shift from implementation to implementation. Any program that bypasses the standard PSL interface, and makes use of private information about this division is no longer insulated from modifications of the definitions of the objects, object managers and the kernel, and the use of such a program may produce unexpected results in the future.

The following is a partial list of the kinds of functions which one may find in the PSL:

- A set of standard interface routines for all operations on the basic Cronus objects. There are two sets of interface routines: those which are designed for use with managers and other asynchronous programs, and which do not wait for the response from an operation; and those which are intended for use in interactive programs, which do wait for a reply if one is expected.
- Functions supporting composite activities, such as writing data on a file specified by a symbolic name.

- Functions supporting the construction of Cronus object managers. These include routines for manipulating UIDs and UID tables, for managing the processing requests and their responses in asynchronous processes, for creating and modifying work-in-process and intentions lists.
- A standard error reporting facility for both asynchronous and interactive processes.
- Sublibraries for message composition, string manipulation, portable input/output operations, and device management.

The PSL is described in detail in Section 2 of the Cronus User's Manual.

6. Interprocess Communication and Messages

6.1. Overview

Cronus presents a set of facilities for the composition of messages and their transmission to provide a systematic communication facility among Cronus processes. There are three parts to this communication support:

- An interprocess communication (IPC) transport facility, based on the object model and object-oriented addressing, provides Cronus primitives for uniform, host-independent communication among processes. This facility, which was introduced in Section 4, is further described in the current section.
- Conventions for passing data using Cronus canonical data types permit messages to be composed without concern for the heterogeneity within a cluster.
- Protocols and conventions for constructing messages used in intercomponent interactions, especially the invocation of operations and the replies.

The Message Structure Library (MSL) organizes these conventions and protocols by providing routines for the composition and examination of messages.

The IPC mechanism of Cronus is built upon the primitive functions Invoke, Send, and Receive. These primitives support the asynchronous communication of uninterpreted data octets among Cronus processes, by means of the abstractions of *sending* to a process or *invoking* an operation on an object.

Messages, the entities communicated by the IPC, may be sent either reliably or with minimal effort. In addition, notions of both a small message which can be carried by a single datagram on the underlying transport mechanism, and a large message which may require an arbitrarily large number of datagrams are supported, although this distinction is hidden by the IPC library routines. Messages may be sent and received all at once or in pieces. The size of the chunk of data manipulated is independently selected by the sender and receiver. Large messages of indefinite size form the basis for interprocess stream communication.

The Message Structure Library (MSL) is used to format messages, but is independent of the IPC. It provides a mechanism for inserting and extracting typed, structured data into a message buffer in a position- and machine-independent manner. Associated with the MSL are conventions, called the Object-Operation Protocol, for the patterns of communication that arise in performing operations on Cronus objects.

The IPC and message structure facilities, and their relationship, will be discussed in the following sections.

6.2. Messages in the IPC

The IPC facility supports two classes of messages: reliable messages and minimal effort messages.

> A message sent reliably will be delivered to the receive queue of the addressed process (or the manager of the addressed object on an Invoke) despite transient failures in the communication substrate. A reliable message will be delivered at most once.
>
> Minimal effort messages are transmitted with whatever reliability characteristics are provided by the communications substrate. The IPC facility does not attempt to provide a sending process with information regarding the disposition of the message.

In both cases, the message is protected by an end-to-end checksum, so if the message is delivered, the content may be presumed to be correct.

The sending process may use minimum effort messages whenever it seems appropriate. The current implementation uses them for all messages sent to a broadcast or multicast address.

Messages may also be categorized by length. A small message will fit into an IPC packet throughout the cluster. The maximum size of a small message is implementation dependent, and in the current system is about 1500 bytes. A large message may have a length set at the time the message is initiated, or the length may be indefinite. Minimal effort messages are constrained to be small, while reliable messages may be small or large.

A large message may be of any size, although they are generally larger than the small message limit, and the PSL automatically selects a small message for messages below the limit and a large message for a message above the limit

Messages of indeterminate length support Cronus streams, which are uni-directional data channels between a source object (sender of the message) and sink object (receiver). Cronus streams are used to interconnect processes with devices and with other processes. Although data flow on the stream is unidirectional, the implementation of a stream involves transmissions in both directions: from source to sink containing data, and from the sink to source containing flow control and synchronization information.

One objective for the IPC facility is to minimize the distinction between small and large messages. In particular, the content and structure of the information contained in a message, and any information about a message that is delivered to a recipient (e.g., size, source, etc.) is independent of its transmission characteristics. The sender of a message indicates whether or not the message is to be transmitted reliably, and its length, if it is of bounded length. The receiver need not be concerned with these characteristics of the message.

6.3. Programming Interface

The programming interface for the IPC provides facilities needed to invoke operations on objects, send messages to processes, and receive messages from clients. Many application programs will be written in terms of higher level routines which may be found in the PSL. The interface described in this section is primarily of interest to systems programmers who are developing and maintaining object managers and PSL routines.

The interface provides direct support for the Cronus primitives (Invoke, Send, and Receive), for the full range of message types (*reliable small*, *minimum effort small*, and *reliable large*), and for various buffering strategies that the sending or receiving process might wish to adopt.

When a process invokes an operation on a Cronus object, it uses the PSL function Invoke; when the message is transferred by the Send primitive, the process uses the PSL function Send. In either case, the process indicates the size of the message being sent, whether it is to be sent using reliable transmission, and points to a buffer which contains the information which is currently available for transmission. The buffer may contain the entire message or any portion thereof. The IPC accepts the information for transmission, and returns a small integer, called the message handle. If there is more information to be sent, a new buffer is given to the SendMore function, along with the message handle. Finally, the message is completed by applying the LastSent function to the message handle.

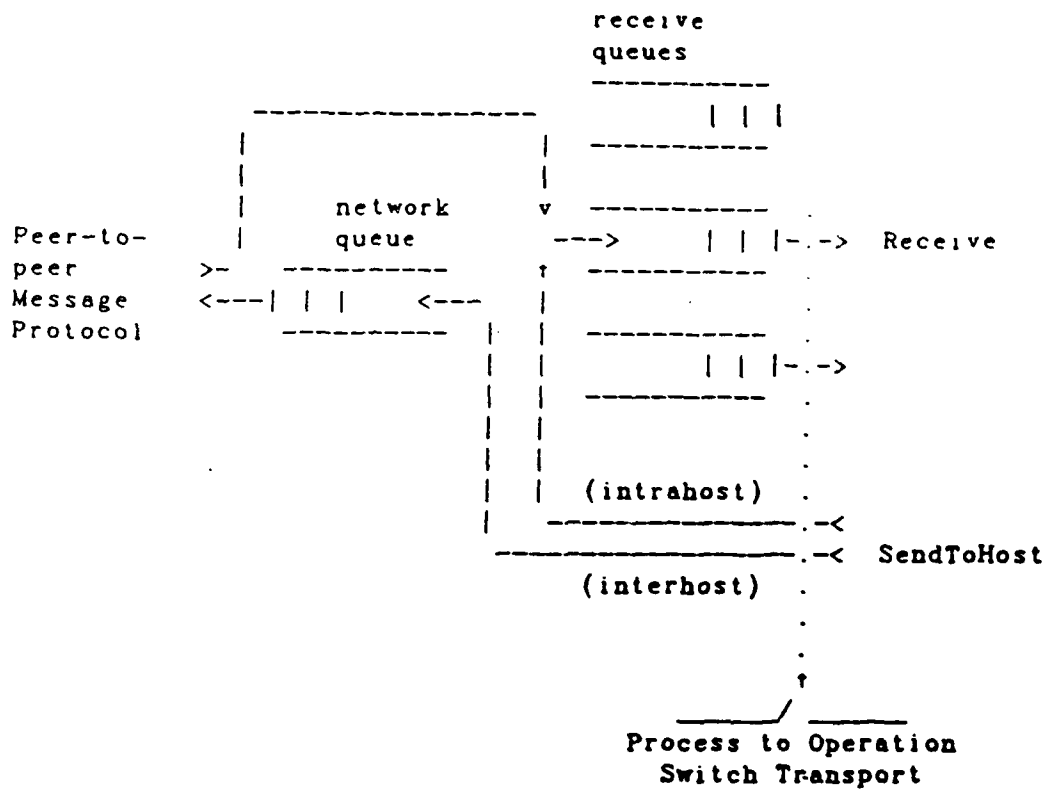
The operation switch on each Cronus host provides buffering for messages and synchronization between Cronus processes. Buffering and synchronization are closely related, because buffering in an intermediary influences the synchronization points between processes.

The sending functions accept the message if it can be queued somewhere within the IPC mechanism. It can be in a host-dependent transport mechanism between the process and the operation switch (see Figure 1), on the "receive queue" of a Cronus process (if it is an intrahost message), or on the "network queue" of messages waiting to be transmitted (if it is an interhost message). If the message cannot be queued immediately, it is refused by the IPC, and the sender is responsible for any required recovery.

Even if the message is accepted, the IPC does not report that the message has been delivered or that delivery can be assured. The only way the sender can be assured that a message has been received by it is to wait for a reply from the intended recipient. Cronus managers respond with at least a ReplyCode whenever an operation is invoked on an object. User processes should normally observe a similar protocol, since lower level protocols cannot assure delivery of messages.

The receive queues are maintained in FIFO order; the network queue is a group of FIFO queues, one per destination host or process. Entries on the receive queues are delivered to client processes to satisfy Receive requests, and entries on the network queue are transmitted to remote operation switches, where they are placed on the proper receive queues.

When the receiving process is prepared to process new data, it executes the Receive or ReceiveMore function. Each new message is started with Receive, and if the entire message is not available, or cannot fit into the buffer that has been given to Receive, more of the data can be read with ReceiveMore. Both functions return immediately with the data, if any, that is available.



Schematic of the Operation Switch
Figure 6.1

The buffering strategies in the two communicating processes may be different. The sending process can, for example, send the entire message in one piece, and the receiving process may choose to receive it a chunk at a time.

The IPC also provides functions which give the client control over the message queues, the basic timeouts which control error handling, and the processing of asynchronous events. These functions include:

- WaitForChange suspends the process until an interesting event occurs. Typically, this will be the arrival of another message or more data for a message which has been partially received. Other interesting events include timeouts and events which are unrelated to the IPC mechanism

- AbortMessage deletes a message from the queue without completing processing (either send or receive).
- SetDefaultTimeout adjusts the standard timeout for the process.
- MsgQueueSize tells how many messages are waiting for processing, including any partially received messages.

6.4. IPC Implementation

The implementation of the Cronus IPC can be described at two levels. There are some elements of it which are generic; the structure of the implementation must support those facilities which clients expect of it. These include the overall issues of buffering, synchronization, and reliability, for example. At the second level, there are specific decisions about how the initial implementation will be constructed. Future implementations of Cronus may choose to do things in a very different way. For example, the current implementation uses the DoD standard connection protocol, TCP, to implement reliable message transport. Future implementations may use a different reliable transport mechanism.

Cronus IPC supports three types of messages:

- small, minimum effort messages;
- small, reliable messages; and
- large, reliable messages.

Neither the protocols used nor the structural requirements of the implementation specify the division of responsibility between the operation switch and the PSL for these various classes of message. In fact, the division might be made differently in different hosts in the same cluster. The transport mechanisms used in the current implementation are shown in Table 6.1.

Small, minimal effort messages are sent from Source Operation Switch to Destination Operation Switch by means of IP datagrams using the standard User Datagram Protocol (UDP). Receipt of an IP/UDP datagram by the Destination Operation Switch is not acknowledged.

On receipt of a datagram, the Destination Operation Switch determines if the enclosed message should go to a local object or process. If so, it places the message on the receive queue of the object manager or process.

Cronus transmits small, reliable messages from Source Operation Switch to Destination Operation Switch over a TCP connection. Although TCP provides services not required for small reliable messages (e.g., strong sequencing, reassembly), we find that the overhead they impose has not made the performance of the IPC unacceptable. If this were the case, we would develop a reliable small message protocol (RSMP). RSMP would perform the following services

TYPE OF MESSAGE	TRANSPORT MECHANISM
Small, minimal effort	IP - Operation Switch <-> Operation Switch
Small, reliable.	TCP - Operation Switch <-> Operation Switch
Large, reliable.	TCP - One connection per large message, connection establishment initiated by an Operation Switch to Operation Switch interaction, but connection may be in the Operation Switch or the PSL, at the discretion of the host implementation.

Message Transport Summary
Table 6.1

- Provide receipt acknowledgement.
- Provide for retransmission.
- Perform duplicate detection and elimination.

As with small minimal effort messages, upon receipt of a message the Destination Operation Switch determines which local object manager or process should receive the message and places the message on its receive queue.

Large messages are implemented through a TCP connection for each message. There is an interaction between the source and destination hosts to establish the TCP connection. When the message has been transferred, the TCP connection is closed.

The following steps are used to establish a new TCP connection to carry a large message between two processes:

1. The source host selects the port to be used for the TCP connection, and puts its end of the connection into the listening state.

2. The Source Operation Switch sends a StartLargeMessage message over the Operation Switch to Operation Switch TCP connection. This message specifies the destination, the port for the TCP connection, and perhaps the first part of the message.
3. The Destination Operation Switch places the message on the receive queue of the object manager or process.
4. When the destination process executes a Receive and finds the first part of a large message, any data sent along with it is delivered. The destination host selects a port for its end of the TCP connection, and uses the TCP port supplied within the StartLargeMessage message.
5. After the connection is established, the source host will use it to pass message data to the destination host.
6. After the source process sends the last chunk of data in the large message, the TCP connection will be closed.

This discussion does not specify whether the Operation Switches or the client processes are responsible for managing the connection that carries the bulk of the message data, nor whether the Operation Switches or client processes are responsible for actually using the TCP connection to send and receive message data. These implementation decisions may be made differently for each host type.

6.5. Object Operation Protocol

The Operation Protocol (OP) is used by the PSL whenever operations are invoked on Cronus objects. There are three basic message types in this protocol: Request, Reply, and InProgress. All of the messages in the OP are marked as belonging to the operation protocol, and each is marked with its basic type. Messages arising from one Request normally contain the same Cronus unique number called the operation identifier. A Request message also contains the operation name and a Reply message contains a standard reply code. These are the minimal contents of the messages; they also contain additional, operation-specific information.

The simplest message protocol involves one Request message generated by a client, and one Reply generated by an object manager in response.

We distinguish between a simple operation and a compound operation. A simple operation has a single operation name and operation identifier. Any manager process, in the course of acting upon a Request may invoke one or more new (simple) operations by sending Request messages. A compound operation is the aggregate of all simple operations arising from or caused by the invocation of one simple operation. Normally, all of the suboperations will complete before the initiating simple operation completes. Each of the simple operations has its own operation identifier, so a process may invoke several sub-operations in parallel.

Sometimes a manager cannot complete the processing required for an operation; for example, a request for a catalog lookup may be satisfied only by the cooperation of catalog managers on two hosts. The manager may then either:

- perform as much processing it can, and send a Reply that is marked Incomplete; or
- elect to complete it using sub-operations, which follow the same pattern as requests, and send a Reply when the operation is complete.

If the manager chooses the first of these alternatives, it can often send the text of the message that the client needs to send to the other manager as part of the Reply. The client can complete the operation by invoking another simple operation.

It is desirable for a Cronus process to be able to query the status of a compound operation. The operation identifier of the original request is used as a global identifier for each suboperation. Since this identifier is included in the Request messages of all simple operations it causes, the managers acting on suboperations can respond to a status query keyed to the initiating identifier.

6.6. Message Structure

The primary design goal for the Cronus message structure is the regularization of *control traffic*. Control traffic includes requests for operations to be performed on objects, replies generated by operations, exception notices, and messages needed to coordinate distributed object managers. Control messages are usually short (tens to hundreds of octets). Because performance is a major issue, messages should be compact, and efficiently composed and parsed.

A message structure can be evaluated in a number of ways. A discussion of evaluation criteria, and an application of these criteria to a number of well-known message structures may be found in [BBN 5261]. As a result of that analysis, a standard Cronus message structure was formulated. It has the following characteristics:

- Messages are self-describing, so the fields may be identified by name rather than by order. This simplifies the parsing of messages, at the cost of transmitting the identifying information.
- The conventions rely only on features that are available in many programming languages. This improves the portability of the implementation, at the cost of increasing the cost of a single implementation.
- The need to define new data types, which are treated in the same way as the pre-defined types, is explicitly recognized. This is consistent with the general philosophy of Cronus design.
- Name and data type fields are compactly coded, and efficient programming interfaces are provided, while the overhead of a general message format is held down. These all contribute to good system performance.

The *Message Structure Library* (MSL) is a collection of functions that is part of the PSL; these routines fall into three classes:

- application interface functions,
- data translation functions, and
- structure manipulation functions.

The application interface procedures construct the message in an *external representation*, which is machine independent, using the data translation and structure manipulation functions. This data structure can be transmitted from one process to another, and subsequently parsed by MSL procedures at the receiving process. A summary of the functions and a cross reference to detailed discussions of them may be found in Cronus User's Manual, in the article *MSL* in section 2.

The Cronus external representation is based on key-value pairs, where the key is a conventional name that is stored with each data value. The key indicates the meaning of the value. The value, in turn, consists of a data type indicator and the actual data. Including the type indicator assures us that we can move the data from one Cronus host to another. The *internal representation* of the data may differ at the sending and receiving hosts, but it is always transmitted in a canonical form, along with its type [Herlihy 1982].

A canonical type is either an *atomic* or *composite* type. An atomic type, such as boolean or signed 16-bit integer, defines a set of primitive data values. A composite type, such as an array or record, has substructure defined in terms of other canonical types.

Keys are coded as short (16-bit) integers, but values can vary in length from one octet to many thousands, and are not restricted in form, and may be built from simple or composite data types.

Most IPC messages passed among managers or between processes and managers use a high-level protocol called the Operation Protocol (OP). OP is based on a set of well-known keys which are used for handling operation invocations and responses. The definition and use of canonical types is described in much more detail in BBN Interim Technical Report #6 [BBN 6183].

7. Authentication, Access Control, and Security

7.1. Introduction

The goals of the Authentication and Access Control facility are:

1. Prevention of unauthorized use of Cronus and unauthorized access to DOS maintained data and services.
2. Preservation of the integrity of the system and its components against intentional insertion of unauthorized components.
3. Support for a uniform user view of access control to the resources and functions provided by Cronus.
4. Survivable authentication functionality

The design of the access control and authentication facility assumes that systems in a Cronus cluster are all in a single administrative domain. There are a three broad classes of hosts within the cluster:

- hosts dedicated entirely to Cronus system functions and not user programmable;
- hosts supporting user applications using tamper-proof multiple protection domains (trusted multi-access hosts); and
- hosts supporting user applications without secure multiple protection domains (single-user workstation hosts).

We assume all hosts supporting dedicated Cronus functions and multiple user protection domains are physically secure from tampering. Workstations may not be completely physically secure, but have at least a tamper-proof component. At minimum, this component is in the local network address insertion and reception function. It could, however, be higher up in the workstation system: in the virtual local network internet address insertion and reception function; in the object system process-unique identifier insertion and reception function; or even higher. In this sense, all user-programmable hosts support multiple protection domains (user and system), although in the limiting case, the "system" domain may simply be a piece of network interface hardware. Since we are not aware of any workstation systems meeting this requirement, we assume future product packaging changes. There seem to be two viable positions to take regarding the assumptions on these changes.

1. Assume only an absolute minimum, that a single low level "address" can be protected.
2. Allow the set of protected functions to grow as needed to conveniently interface the workstation in a manner as similar as possible to multi-access systems.

The extreme solution to the second approach could be an access machine for each workstation, although other solutions are also possible. For our current work we will assume the second approach, planning only for an arguably insecure implementation directly within the workstation.

The network (cable) itself may also not be totally physically secure. While parts of it can be expected to be secure (e.g. within a secure machine room), other parts can be expected to be exposed to unauthorized connection.

7.2. The Cronus Access Control Concept

7.2.1. Decomposition of the Access Control Problem

The basis of access control in Cronus is the ability of Cronus to reliably deliver the address of a sender of a message (or invoker of an operation) to the receiver of the message. The Cronus communication subsystem is implemented so that this is true. That is:

for IP and Virtual Local Network:

> If the sender is within the Cronus cluster, the internet host address of the sender is reliably delivered to the receiver. If the sender is not within the cluster, a non-cluster internet host address is delivered to the receiver, which can be interpreted by the receiver as indication that the authenticity of the sender's address might be suspect.

for the Cronus IPC/object system:

> The UID of the sending or invoking process is reliably delivered to the recipient of the message.

The recipient of a request can decide on the basis of the sender's identity whether or not to perform an operation requested.

For this to be a useful basis for access control, a means for reliably associating some authorization with senders' addresses and process UIDs is required.

One approach is to make static bindings between authorizations and addresses or UIDs. These bindings would be "well-known", such that when a process receives a request from the process with UID_Y it knows that the process is acting under the Z_Authority. This method is used in the ARPANET TELNET and FTP protocols; users assume that the process for sockets one and three are under the authority of the host administration and can be trusted with their passwords. Static bindings are too restrictive to be the sole mechanism in a system like Cronus, although a few static bindings are required for the access control mechanism to work (see Section 7.6).

Dynamic binding is useful when authorities are not all known at system creation time, and when processes are dynamically created. The system must not only support mechanisms to dynamically establish the binding between a process and an authority, but also to dynamically determine the binding from some system entity in a trustworthy manner.

Most Cronus activity is the result of requests initiated by users of the system. Human users are represented by an abstraction called a "principal". If we extend the notion of a principal to include elements of the system, such as object managers, all activity in the system can be thought of as initiated by principals. System elements which are principals are called "system principals". Each Cronus principal (human or system entity) has a unique identifier. Different system principals have different authorities. For example the primal file manager and the printer service are Cronus system principals, neither of which need be authorized for all of the objects and operations accessible to the other.

Access control can be thought of as consisting of the following steps:

1. Identification. Determine the identity of the principal that is requesting a particular operation.
2. Authorization. Determine whether the principal has been authorized to perform the operation.

For example, when an object manager must decide whether to perform an operation, it must know the identity of the principal that is requesting the operation (Identification) and the rights the principal may have with respect to the operation (Authorization).

7.2.2. Authorization

Cronus uses access control lists to support authorization. The access control list (ACL), which is part of the object descriptor, "protects" a particular action. In the simplest case, it is a list of the principals who have authorization to perform the action. When a principal attempts an operation, the list is checked for the principal: if the principal is present the authority to perform the operation has been verified and the operation may occur.

In Cronus this simple idea is extended in two ways:

1. Group identifiers may appear on an ACL, so an entire group of principals can be authorized as a unit, or have its authorization revoked as a unit.
2. A set of rights is associated with each identifier on an ACL. A single list can selectively control a principal's or a group's access to an object for which several operations are defined, such as a file. Rights are abstract, bound to specific operations by the implementer.

An ACL is a list which contains elements of the form:

(id, rights)

where "id" is either a principal (PID) or a group identifier (GID), and "rights" define the principal's or group's authorization with respect to the object the ACL protects. The allowable rights for a particular ACL are dependent upon the type of object being protected.

Users log into Cronus as principals by supplying an appropriate name and corresponding password. A system component called the Authentication Manager maintains records of all principals and groups. Collectively, these records form a User Data Base (UDB). At login time the Authentication Manager expands the membership of a user-specified subset of the access control groups which he is a member. This is a transitive closure computation on the specified list of group identifiers in the user's record. The user's own id, PID, is added to the result of the expansion. The resulting set of principals is called the access group set (AGS) for the process:[9]

AGS = {PID} U Show_Group_Membership_Expanded (GID)
for the default GIDs in the PID record.

The AGS is used in access control checks as follows. When an action protected by an ACL is attempted, the ACL is compared with the principal's AGS. If an entry of the form:

(ID, (..., Right, ...))

where

ID is in AGS, and
Right is required to perform the action

is found on the ACL, the principal's authorization is verified and the action may be performed.

During a session, a user may add and remove identities from the current AGS. To add a group identity, the user must be a member of the added group. Updating the current AGS is accomplished via operations invoked on the Authentication Manager, which causes the update of the current process AGS list. These operations affect a single process however, the new AGS will be inherited by subsequently-created children only.

7.2.3. Identification in Cronus

There are two related identification problems:

1. At the start of each session, the identity of the user must be established.
2. Processes must be able to ascertain the identity of the principal corresponding to the processes with which they interact.

The solution to both problems lies in a set of mechanisms that bind processes with principal ids and group identifiers. These mechanisms depend upon the ability of the communication system to deliver the UID

[9]The basic ideas associated with Access Group Sets have been adapted from similar work at Carnegie Mellon University in the Central File System project.

of a sending process to the receiver of a message reliably.

It is useful to restate these problems into the following terms:

1. A binding must be established between a process and an AGS;
2. There must be a means for a process P1 to determine the binding between another process P2 and its AGS.

When a user approaches Cronus to start a session a process (P1) is allocated[10]. P1 cannot be bound to U (the user's principal identifier) until Cronus establishes the connection via password authentication. Before that happens, P1 is bound to a well-known principal, "NotLoggedIn", which has minimal authorization. One task of the login procedure is to change the binding of P1 from NotLoggedIn to U.

The binding between a principal identity and a process is established by the Authenticate_As operation. The user engages in an authentication dialogue with Cronus, supplying a name and password which is checked against the UDB. If the authentication dialogue succeeds, the AGS for U is computed and a binding is established between P1 and U. A record of the binding

P1, U, AGS

is maintained by the process manager for the authenticated process, to be used throughout the process lifetime. The identity of the user has been established, completing problem I1.

Throughout the course of U's session, P1 and other processes acting on behalf of U attempt actions which require authorization verification by the processes that perform the actions. This is problem I2. Consider a situation in which P1 has requested another process (S1) to perform some action (A), shown in Figure 1.

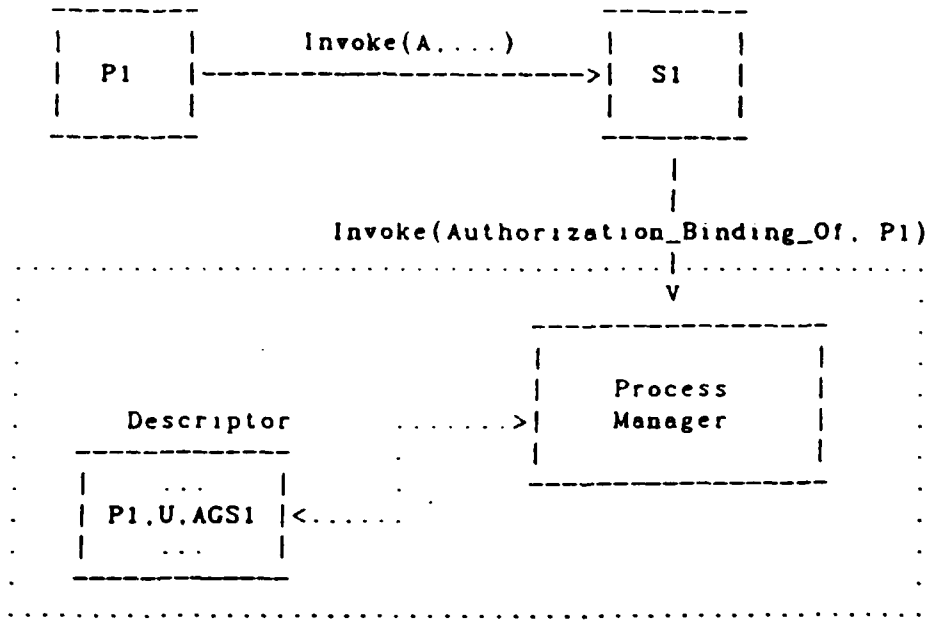
In order to perform an access control check, S1 needs to determine the binding of P1. The identity of P1 is known to S1 because P1's UID was delivered along with the operation invocation that requests A. S1 can obtain the binding of P1 by invoking the Authorization_Binding_Of operation:

Authorization_Binding_Of(P1) -> U, AGS.

Authorization_Binding_Of causes a message to be sent from S1 to the manager for process P1, which returns the bindings for the process to S1.

The login sequence establishes a binding between user (U) and an "initial" process (P1). Bindings are established for other processes created during a user session through inheritance. During a user session, processes created by an authenticated process inherit both the principal identity and the current AGS of the initiating process. Object managers attain their principal identities and access group sets as

[10]Cronus actually uses a more complex process structure to support a user session. However, the following discussion is insensitive to these details, so we use this simple model in our explanation.



Retrieving Access Control Data
Figure 7.1

part of the system initialization phase.

7.3. Access Control List Initialization

A common problem associated with Access Control List mechanisms is the effort required for proper explicit (manual) initialization. In practice, the ACL for a new object can often be automatically predetermined based upon the type of the object, the creator, and the context in which the object is created (primarily the directory in which it is subsequently catalogued). This is the premise upon which the Cronus Initial Access Control List (IACL) mechanism is based.

A list of type-specific IACLs may be associated with selected Cronus objects, currently Principal and Directory objects. The IACLs are manipulated using the standard ACL manipulation operations (ReadACL, AddToACL, RemoveFromACL), distinguished by an optional key denoting the type with which the IACL is to be associated. The IACL mechanism also supports the Cronus type hierarchy: the IACL associated with an ancestor in the type hierarchy will be used if a more specific IACL for the type itself has not been specified.

Cronus Create operations incorporate the following algorithm for initializing the ACL of newly-created objects:

1. A list of "IACL hints" (UIDs of objects potentially having IACLs associated with them) are searched in order for an IACL pertaining to the type of the object being created. The first one found is used. These hints usually reference the Cronus directory where the object will subsequently be catalogued.
2. If no IACL search is specified, or the hints fail to yield an appropriate IACL, the object for the Principal invoking the operation is queried as if it were included at the end of the hints list.
3. If an IACL is still not found, the invoking Principal is given all rights to the object.

There are user commands for setting up, examining and modifying the initial access control lists retained with cronus objects.

7.4. Authentication Manager

The Authentication Manager defines and maintains two types of abstract Cronus objects: CT_Principal and CT_Group. Like other system objects, the CT_Principal and CT_Group identifier objects have symbolic names for convenient human access. Principals are symbolically named from a private name space maintained by the Authentication Manager, which ensures their uniqueness across the entire system. Symbolic group identifiers can be placed anywhere in the Cronus catalog, at the convenience of the creating user.

Operations on objects of type CT_Principal and of type CT_Group are controlled by access control lists. By convention, any legitimate principal can create a new CT_Group object, but only administratively authorized principals can create a new principal. When the system is initialized, it contains at least one pre-defined principal, which is authorized to create other principals.

In the following sections we discuss the design of the objects and operations supported by the Authentication Manager. Section 7.8 discusses how to make the functions of the Authentication Manager survivable.

7.5. Objects Related to Authorization

The object of type CT_Authentication_Data is the user data base consisting of the records for system users and for groups of principals which have been defined in the system.

The object of type CT_Principal is the permanent data base entry that Cronus maintains for each legitimate user. It is the repository for such user-specific data as default priority and other parameters associated with resource management; default modes of behavior (e.g. default working directory); and authorization data. It is expected that new kinds of data will be added to the principal objects from time

to time.

A CT_Principal object can be expected to contain the following data:

- Principal unique-identifier (PID)
- Symbolic name of principal
- Access control list
- Encrypted password
- Direct group memberships
- Direct group memberships to be expanded on Login
- Range of priority service authorized
- Default priority
- Name of default initial subsystem
- Name of home directory for the principal ... (other user-specific data)

The priority data will be used in resource management functions. The default subsystem is the program automatically invoked following login. A home directory is a directory assigned to the principal that serves as the initial current directory for catalog accesses; in particular, it contains additional user initialization data.

Groups (objects of type CT_Group) gather a number of identities for purposes of collectively granting them rights to objects and operations. Any user can create a new group, and place any other principal or group in it. This group can then be placed on an ACL. The access control list for the group object controls modification of the group definition.

A CT_Group object contains at least the following data:

- GID for the group
- Name of the group
- GIDs of the groups of which the group is directly a member
- IDs of principals (PIDs) and groups (GIDs) that are direct members of the group

There are a few special group identifiers. One of these (group world) represents the set of principal identifiers without actually enumerating them anywhere. This group identifier is automatically appended to every AGS computation. Another special group "Wheel" represents an access control override capability used for system maintenance, implicitly receiving all rights to all Cronus objects. Admission to this group is carefully controlled.

A convention has been adopted which effectively supports wheel capability only for objects of a specified type. A process whose principal ID matches the PID of the manager process is automatically granted all rights to all objects managed by that manager. This is useful in handling peer managers. As an example, all file managers are bound to a special file manager principal, and implicitly have all access to all files managed by peer file managers.

7.6. Operations on Authorization Related Objects

The generic operations to create and remove objects, and to examine and modify the object descriptor, ACL, and object status apply to instances of CT_Principal and CT_Group.

The following operation is used during login to establish the binding of the user to the principal UID:

Authenticate_As

The following operations allow processes to control the identities applicable to an authenticated process. They effect only a single process, which may be either the invoking process or another process authenticated to the same principal.

Enable_Access_Group
Disable_Access_Group

The following operations maintain and interrogate the objects of type CT_Principal:

Lookup_Principal
Show_Group_Memberships
Add_to_Default_Group_Expansion_List
Delete_from_Default_Group_Expansion_List
Change_Password

The rest of the data in the principal entry in the user data base is treated as part of the object descriptor. The generic operations which manipulate the object descriptor are used to examine and set these fields.

The following operations are used to inspect and maintain the group identifier objects:

Add_to_Group
Remove_from_Group
Show_Group_Members

The rest of the data in objects of type CT_Group is contained in the process descriptor and is maintained using the generic operations defined on object descriptors.

The access control list of any object, including objects of type CT_Group and CT_Principal, can be set using the generic operations on access control lists.

7.7. Operation of the Access Control Authorization Function

Cronus access control checks the current identity of the accessing agent against access control lists maintained by the service provider. A process is authenticated in a way which binds the process UID to a set of external identities defining the authorizations of the process. These identities, the AGS, are available to any service-providing process. This section discusses the authorization function which is part of the service provider.

In general, the access control steps within an object proceed as follows:

1. The request is parsed to determine the originating process UID and the operation/object requested. The process_UID is trusted because it is added to the message by the operation switch. Universal public privilege for the operation to all objects managed by the manager is first checked, to see if the specific access check is needed.
2. A manager-based cache of process/object authorization pairs for the process_UID is checked for a valid current entry.
3. If there is no corresponding cache entry, the accessing agent's AGS is obtained. This data is also cached but on a per-host basis by the AGS cache manager. If present on the host, this cache manager provides a high performance interface to the Authentication_Bindings_Of function. There is a broadcast-based protocol for alerting AGS cache managers to entries that should be purged. If an AGS cache manager does not run on a host, managers execute the Authentication_Bindings_Of operation directly, and the AGS is not cached. [The per host AGS caching is not yet designed or implemented.]
4. The access control software computes a new process_UID/object authorization entry using the AGS and the access control list maintained with the protected object/operation. The process_UID authorization entry is then put in the manager cache.
5. The process UID object authorization is used to verify permission. If authorized, the operation is passed on to the operation code. If unauthorized, the request is rejected.
6. To allow for the enabling of new access groups, steps 3-5 are repeated in the event that cached AGS fails.

The permission authorization function is accomplished by a set of routines and data structures that we call the "gatekeeper" because of its role as protector of the objects/operations. Gatekeeper functions can be invoked as part of the procedures for receipt of a message, or called directly from the host process.

Access control can be applied to operations on the object set supported by the receiving manager process, or on operations defined by the receiving service. There is a fixed maximum number of access control rights maintained by the gatekeeper software (currently 32) for any object. These rights are represented as positions in a bit vector associated with both the identity it authorizes (principal identifier or group identifier) and the object it controls.

7.8. Host Registration

The lack of physical security for various parts of the system presents problems for the access control subsystem. Since the network cable may be accessible to tampering, the network might be tapped. An outsider could then inject or inspect packets under an assumed network address. A workstation might pose as the site of a trusted manager. We can use administrative authorization to alleviate these problems.

Encryption of all local network traffic is a form of authorization. It can remove the threat of tapping for either listening for or insertion of packets. Providing the host with the encryption/decryption key is administrative authorization to participate in the Cronus cluster. If a host can communicate at all, it can be considered an authorized host. Because encryption/decryption is isolated in the communication interface, it can be added transparently at any time. While communication encryption can be thought of as part of the Cronus design, it will not be part of the initial implementation.

Since workstations may be treated specially for some access control decisions, system configuration registry could be the source of such identification. In addition, the undesirability of tightly controlling responses to broadcast Locate operations, makes the registry useful in determining the authenticity of the respondee. A configuration registry enumerates all of the authorized system hosts, and the system services (Cronus functions) which they have been authorized to run.

One secure way to make the registry service available is to support it on one (or more) well-known Cronus hosts (i.e. hosts at a well-known internet addresses, say host No. 1, ...). The configuration data can then be obtained with an Invoke On Host to the well-known hosts using the logical name for the service[11]. The cluster configuration service would support the following functions:

Show_Configuration_Hosts
Set_Configuration_Hosts

Standard access controls apply, with Show_Configuration_Hosts being universally allowed, while Set_Configuration_Hosts limited to a system administration group.

[11]Since this function is often used to determine the veracity of responses to the Locate operations, it can not safely use Locate to find out where configuration managers are running.

7.9. Survivable Authorization Design

7.9.1. Objectives

The authentication function and evaluation of the current AGS are critical parts of the operation of Cronus. These functions must be available at all times or Cronus cannot operate effectively. Our objectives in providing survivability in Authentication are:

a. A Cronus user should, under reasonable failure patterns, always be able to gain access to the system.

b. The current value of the process-AGS binding should be available whenever a process is able to request services from object managers.

c. A less important but desirable objective is that a client be able to continue to perform maintenance operations on the principal and group objects despite failures of hosts supporting these functions.

To meet objectives (a) and (c), we must replicate the Authentication function. To meet objective (b), we must maintain the bindings in a replicated fashion, or keep them close to the process to which they refer, so that the bindings are available when the process makes requests of other Cronus managers.

7.9.2. Observations

The authentication function is a global DOS function supported on a GCE which is expected to be up most of the time. Because these services are simple, the host hardware and software should be stable, increasing its availability. Since the GCE is relatively inexpensive, it is also feasible to stock a spare.

The authentication function is based on maintaining two related types of objects. The data bases which the Authentication Manager maintains to support the principal and group objects are not large. The principal data base is estimated to be no larger than 1000 users, with an average entry having around 1000 bytes of data. The group data base might have 2000 entries, averaging 300 bytes of data. This is less than 2 MBytes of data, and can easily be accommodated on a GCE.

The processing demand on Authentication managers is not expected to be large. Aside from initial authentication and group expansion, which occurs typically once per user per session, other operations are infrequent. New users and groups are occasionally created and the associated data bases occasionally displayed and updated. A single GCE appears easily capable of handling anticipated processing requests.

Performance and size considerations do not seem to require more than a single GCE per cluster. Survivability is the primary motivation for replicating the authentication manager. Our approach is to maintain completely replicated data bases on two or more GCEs.

Of the operations performed by the Authentication Manager, the one of most concern for survivability is Authenticate_As, which is a read-only function. This is also true of a number of other AM operations (Lookup Principal, Show Groups Expanded, etc.). Synchronization of multiple authentication managers is not required to complete these operations.

Some AM operations do modify the authentication data (e.g. Create new principal, Modify User Parameters, etc.). These require synchronization among Authentication Managers for consistency. However, because these operations are relatively infrequent and have simple semantics, a simple approach to synchronization which ignores maximizing concurrency will suffice. We designate a primary Authentication Manager as a single point of synchronization. This method is backed up by an alternate procedure if the primary site is inaccessible. A complete description of our approach follows in the next section.

In the current implementation, each process has a process manager on the same host. The process-AGS bindings are maintained by the process manager in the process descriptors for these processes. During host outages when a manager is inaccessible, so too will be the process it manages. There is no need to maintain the process-AGS binding any more reliably than we maintain the process reliability. As some later point, we will address issues of process survivability. We can then naturally think in terms of replication of process descriptor data (including the current AGS) as part of the reliable process concept, and need not address it separately.

7.9.3. Approach

Fully redundant copies of the authentication data bases are maintained at more than one Cronus host. This means that, ignoring synchronization, an operation can be completed at any site which maintains the data base. We expect that two operational authentication sites will provide sufficient availability for most applications of Cronus.

A spare GCE could be integrated into the system if one of the dedicated hosts needs to be taken off-line for any extended period. This minimizes the time during which there may only be a single Authentication site functioning. The new host integration protocol first involves transmission of all of the existing objects. When the object transmission is complete, the new manager retrieves the change log and incorporates any updates. The final step before assuming operational status is to coordinate with any on-going activities.

Each operation on authentication data objects is an independent transaction, so that there is no linkage between any two operations. The operations either reference the identified objects (read operations) or modify the identified objects (write operations). Read operations require no synchronization or concurrency control between Authentication Managers. Any Read operation can be handled by any available authentication manager. Some read operations have side effects which do change the state of other system variables (e.g. Authenticate_As modifies the current process AGS in its process descriptor) but these are idempotent operations so repeating them at distinct sites as part of error recovery is not harmful.

Write operations, on the other hand, require synchronization among the Authentication managers to preserve the consistency of the data with respect to concurrent updates. To do this one AM is chosen as the primary site. The designation of which AM is primary is found in the configuration data base for the system. Clients as well as other AM processes can consult this data base to find the primary site. The primary site remembers its role and will respond to broadcast request to identify itself in case the configuration file is inaccessible.

All Write operations are initiated with the Primary AM, which serializes the modifications to the database. The primary AM records the modification in a change log by appending a change record to a multi-copy reliable file. After logging the request, it updates it own data base, and informs other operational AMs of the change. If all AMs are running, the data bases are again synchronized after each one incorporates the update. When an AM is restarted, it processes the change log to incorporate changes made to the data base in its absence before it will accept new requests. Multi-copy files are used for change logs to avoid single host failure reintegration dependencies.

This approach raises two issues:

a. What, if anything, should we do about read/write synchronization for read operations that may be processed by a non-primary AM while the corresponding object is undergoing modification by the Primary AM?

b. What, if anything, should we do when a modification is requested and the primary AM is inaccessible?

To answer question (a) we first observe that not only is the data changed infrequently, but much of it is particular to a single Cronus user, and hence concurrent read and write access is quite unlikely. Furthermore an old copy of just modified data is almost never harmful. The behavior is similar to a race condition between independent accesses to a single copy data base. Thus our approach to Read/Write synchronization is to do nothing.

There are many possible answers to question (b). One approach is to do nothing, and reject these operations temporarily until the primary AM is brought back on-line. Since modifications to authentication data are not critical to the operation of the system, the major effect of this is inconvenience because we will need to repeat the operations at a later time. A simple mechanism which avoids this uses the lock on the change log file as a tool for serializing updates from any of the available AMs. In this scheme, when the primary AM is inaccessible, any AM can initiate the update if it can first lock the change log. It then informs the other operational AMs of the change. When the primary comes back, it integrates the changes it has missed before assuming primary update responsibility again.

8. Symbolic Naming

8.1. The Cronus Symbolic Name Space

Cronus has a global symbolic name space with the following properties:

1. Cronus symbolic names are location independent.

   - A name for an object is independent of its host.
   - A name that refers to an object can be used regardless of the location from which it is used.

2. Cronus symbolic names are uniform: common syntactic conventions apply to names for different types of objects.

The symbolic name space is constructed upon a hierarchically structured tree. The tree contains nodes and directed labeled arcs. There is a distinguished node called the "root". Each node has exactly one arc pointing to it, and can be reached by traversing exactly one path of arcs from the root node. Nodes in the tree represent Cronus objects which have symbolic names. Links provide an overlaid structure based on symbolic pointers which provide a name space which is a network, so a node may be reached by more than one path.

Non-terminal nodes (those from which arcs may originate) are called directories. Each labeled arc corresponds to a catalog entry. The label for an arc is called an "entry name".

The complete name of a node, which is the symbolic name for the object, is formed by concatenating the labels on the arcs traversed on the path from the root node to the node in question, separated with the character ":". In other words, the syntax for a complete name is:

: { x : }* y

where "x" and "y" are arc labels, the "{","}" brackets indicate optional presence, the ":" is a punctuation mark to separate name components, and "{ s }*" means zero or more occurrences of s.

It is also possible to name nodes relative to a directory. Such a relative name is formed by concatenating the labels on the arcs traversed on the path from the directory in question to the node. The syntax for a relative name is:

{ x : }* y

Conventionally users have a standard directory for relative path names. This is known as the user's "working directory".

The most common types of cataloged objects are the various kinds of files, but any other object may be cataloged. Some conventions have been adopted; for example, there is a *:printers* directory which contains the symbolic names for printers on the system. These conventions are not enforced by the system, and any object may be entered into any directory (assuming appropriate authorizations) at the convenience of the user.

There are certain special object types which are used in support of the catalog itself, including:

- Directories: A directory object (type CT_Directory) is a non-terminal node in the catalog tree.

- Symbolic Links: The catalog entry for a symbolic link (type CT_Symbolic_Link) identifies another point in the symbolic name space called the link target. These objects are stored in the catalog itself. Links are cataloged as terminal nodes in the name hierarchy tree. Links are handled specially within the Lookup operation.

- External linkages: An external linkage (type CT_External_Link) is an object which implements access to another name space. External linkages are cataloged as terminal nodes in the name hierarchy tree. External linkages permit users to refer to non-Cronus objects directly from the Cronus name space. For example, an external linkage might be used to give a file directory on a Cronus application host a Cronus symbolic name.

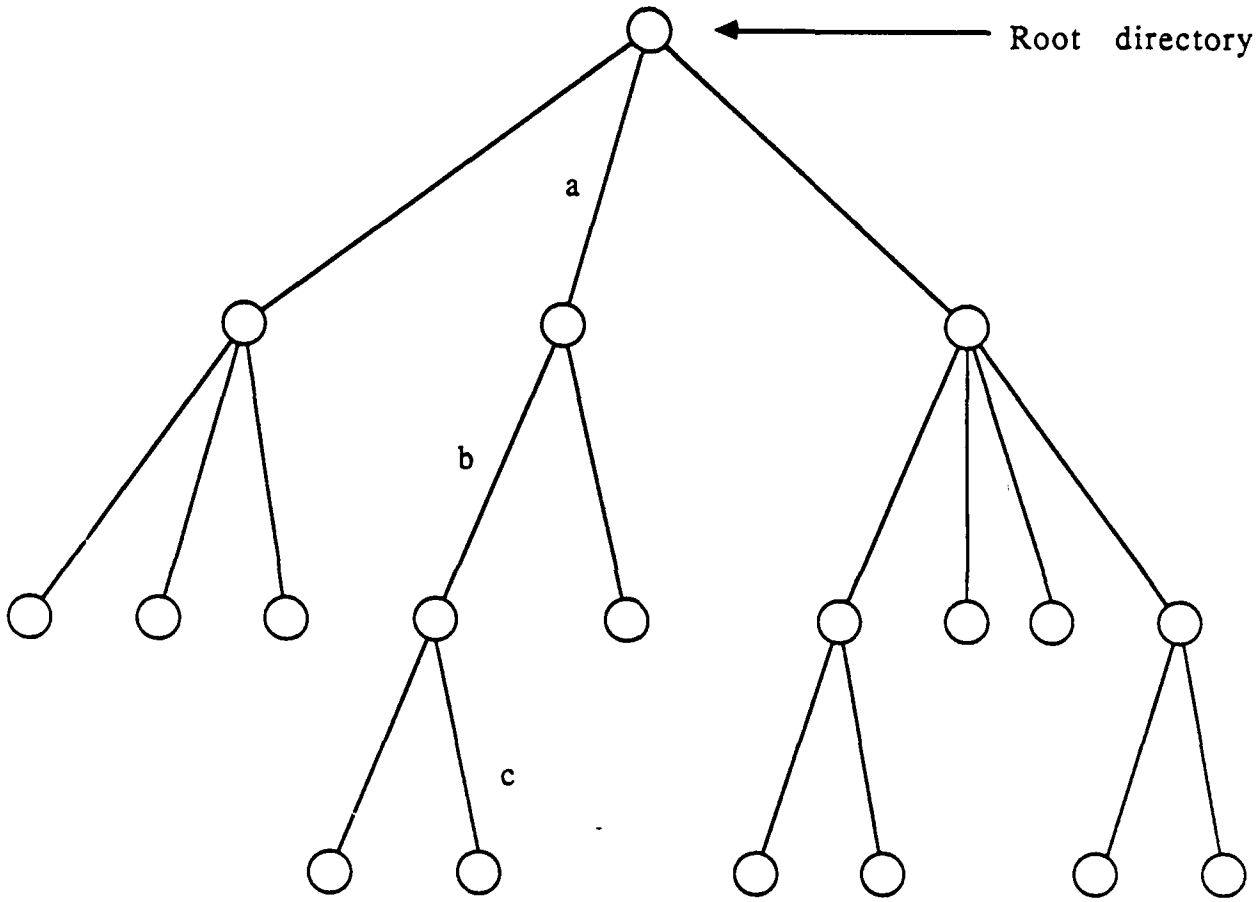
For some object types it is useful to be able to think of a collection of the objects as a sequence of "versions" or "revisions" of the same logical object. The Cronus Catalog implements a version feature for catalog entries. The create catalog entry operation permits the same name to be entered into a directory more than once. Each copy of the entry has a distinct version field and points to a different object. However, all objects pointed to by different versions of the same entry name must be of the same type. The first time a name is entered, the result will be version 1 of the object. Subsequent entries of the same entry name will result in successively higher versions of the object. All of the catalog operations which take a name parameter will allow the specification of a version number as well.

The catalog managers provide routines that can scan through the catalog and return catalog entries for names that match a specified pattern.

The *create catalog entry* operation can be used to simply establish a symbolic name for a Cronus object of any type except a symbolic link or external linkage object. These types of entries are inserted in the catalog when they are created (since other objects need not be named, the creation of the object and naming of the object are distinct operations). In a sense, these objects are special in that they must have a symbolic name in addition to a UID.

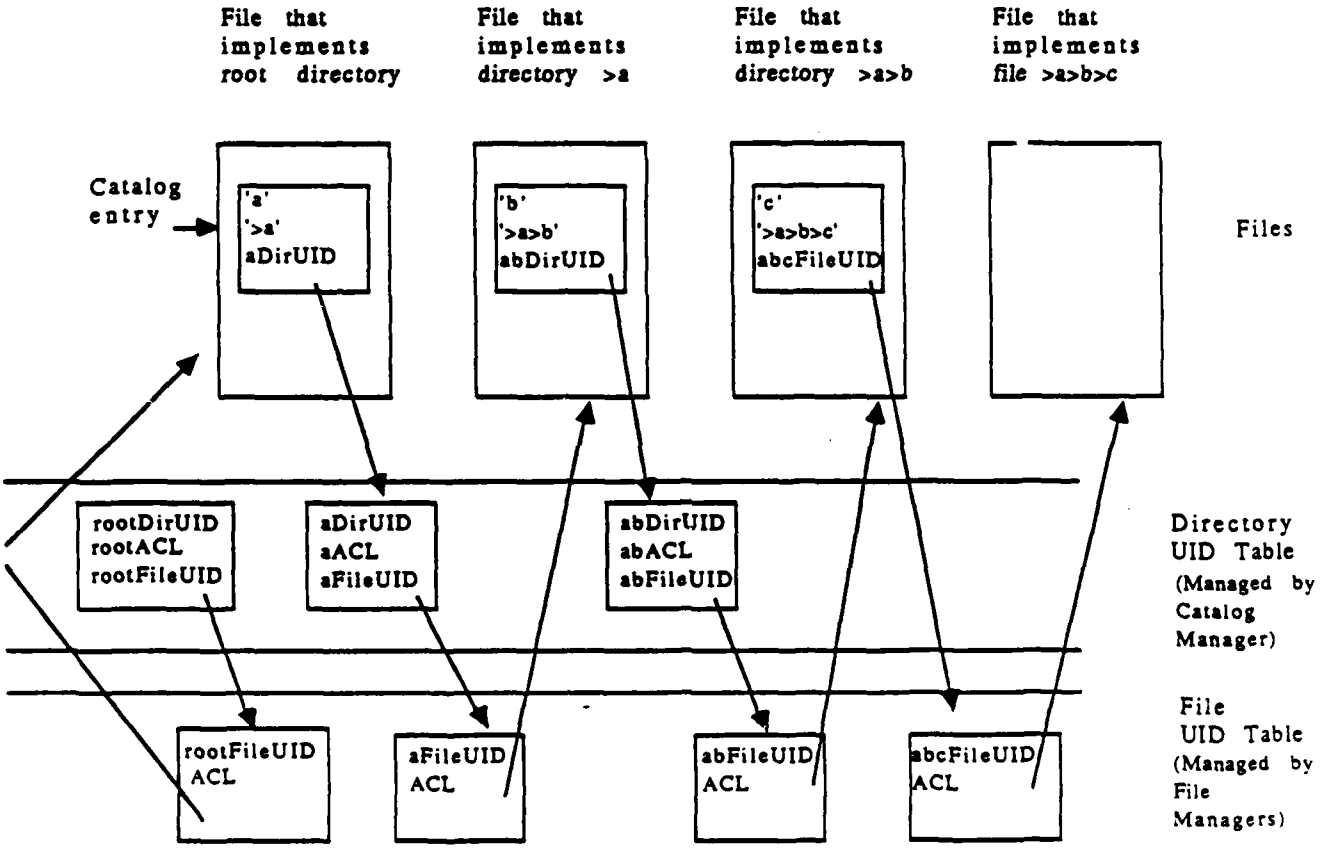
Figure 8.1 shows a relatively simple symbolic name tree and Figure 8.2 shows part of the underlying directory structure that corresponds to the part of the tree that contains the name :a:b:c.

When a *lookup* operation is invoked, the catalog manager interprets a complete Cronus symbolic name by starting at the root directory. The UID of the root directory is well-known. The catalog manager processes a name component by searching the current directory for a matching catalog entry. If it finds a matching entry and there are no more name components, the lookup is complete and it returns the catalog entry. If it finds a matching entry and there are more name components to interpret, the



Catalog Hierarchy
Figure 8.1

Implementation of Cronus Catalog



Implementation of Cronus Catalog
Figure 8.2

entry must be for a directory, symbolic link, or external linkage, or else the lookup ends in failure. If the entry is a directory, the catalog manager continues the lookup by obtaining the UID for the directory from the entry and then using it to interpret the next component.

Interpretation of a relative symbolic name is handled in the same fashion, differing only in where the lookup starts. For a relative name, the catalog manager starts its search at the starting directory parameter of the lookup operation.

Symbolic links encountered during lookup are handled in a special manner. When a link is encountered, a new name is formed by substituting the link target, which is a complete Cronus symbolic name held in the catalog entry, for the portion of the symbolic name evaluated so far. The lookup operation then resumes by interpreting this new name. Links can be thought of as macros which are expanded during the lookup operation.

A parameter of the lookup operation controls whether links are to be expanded. If the parameter specifies that links are to be expanded, the substitution of link targets during the lookup operation occurs. If the parameter is set to prevent links from being expanded, the lookup operation terminates when a link is encountered. In this case, the lookup operation will be considered successful if the name has been completely evaluated. Otherwise it will be considered a failure.

8.2. Structures Used in the Catalog

8.2.1. Directories

Directories are Cronus objects which contain lists of catalog entries. All operations on the catalog or on catalog entries are invoked on directory objects. This includes the root directory which is special only in that its UID is well known. In general an operation on a catalog entry may be invoked on any directory in the path name; specifying the relative entry path as a request parameter.

Since directories are Cronus objects they have many standard properties. Catalog Managers manage directory objects and perform all the generic object operations on type CT_Directory. In particular, access control in the catalog is accomplished through the use of standard Cronus mechanisms on directory objects. Thus, a user may lookup a path name if he has the necessary rights on each directory component in the name.

8.2.2. Catalog Entries

A catalog entry is not a Cronus object as it has no UID. It is object specific information associated with a directory object and consists of the following fields:

- Entry name and version number;
- UID for the object;
- A host hint for the object; and
- Type-dependent information.

Type-dependent information for objects of type CT_Symbolic_Link and CT_External_Link is discussed below. For objects that are not part of the Cronus catalog, everything that can be known about an object is maintained by (or can be obtained from) the manager for the object. That is, no type-dependent information is maintained in the catalog.

8.2.3. Symbolic Links

A symbolic link is a may be thought of as a dummy object maintained by the catalog manager. Although it has a UID, operations may not invoked on a symbolic link. The UID is used only to distinguish it from other catalog entries. A symbolic link consists of the same fields as any other catalog entry; however the type-dependent information consists of the complete symbolic name for the link target. The catalog manager uses this information when performing lookups.

8.2.4. External Linkages

An external linkage is much like a symbolic link. It is distinguishable from a standard catalog entry by the type field in its UID, which is set to CT_External_Link. The type-dependent information in the external linkage specifies the data about the external linkage. It a Cronus interpretable designator for locating the other name space and a symbolic name that is interpretable in that space.

8.3. Catalog Operations

8.3.1. Objects of Type Directory

Operations on the Cronus symbolic catalog are performed on object of type CT_Directory. Currently the following operations are defined for directories:

AddToACL
Create
CreateEntry
CreateExternalLink
CreateSymbolicLink
Dereplicate

DumpLog
DumpObject
Locate
LockObject
Lookup
LookupWild
ModifyEntry
ReadSysParms
ReadUserParms
Remove
RemoveEntry
ReportStatus
SetLoggingLevel
UnlockObject
WriteSysParms
WriteUserParms

Most of these are generic operations which are inherited from parent object types CT_Object and CT_ReplicatedObject. See section 4 of the Cronus User's Manual for more about inheritance. Only *CreateEntry*, *CreateExternalLink*, *CreateSymbolicLink*, *Lookup*, *LookupWild*, *ModifyEntry*, and *RemoveEntry* are unique to the catalog. The remainder of this section describes these operations.

CreateEntry, *CreateExternalLink* and *CreateSymbolicLink* are used to create entries in a directory. The second two actually create special entries: external linkages and symbolic links. If specified entry already exists these operations create a new version of the entry. The version number may be specified, but ordinarily the next highest version number is given to the new entry.

Lookup is used to look up a catalog entry given a path name. All the information associated with the entry is returned. By default the highest version of the entry is returned, but the version number may be specified. *LookupWild* performs a catalog lookup using Cronus wild card conventions, and returns a list of all the entries which match the specification.

ModifyEntry changes any of the parameters associated with a specific catalog entry. *RemoveEntry* removes an entry. Once again, these operate on a single version if there are more than one present. Default rules apply if no version number is specified.

8.3.2. Access Control In The Catalog

Access control is performed in the catalog by using the standard Cronus access control mechanisms on objects of type CT_Directory. When a user wants to perform an operation on the catalog he invokes the operation on the appropriate directory. If the manager of that directory determines that the user has the apropriate rights the operation is performed. If not the operation fails.

The access control problem is slightly complicated by the fact that path names in Cronus can reference several directories. If a request look up the path name ":animals:mammals:cat" is invoked on the root, the catalog manager must traverse through the directories ":" and ":animals" before it can look up "cat" in the ":animals:mammals" directory. The catalog managers deal with this by doing *Lookup* access control checks on each directory in the path.

It should be noted that access restrictions on a object's entry information is not related to access restrictions on the object itself. The catalog is generally used to look up object UIDs so that operations can be performed on those objects. Individual object managers perform their own access control on their objects. Therefore, it is possible to be denied look up access on an object name but still have all rights to manipulate the object itself, and it is possible to be denied all rights to an object for which one has look up access to its name.

8.4. Catalog Implementation

8.4.1. Introduction

The following implementation issues are discussed below:

1. the manner in which client processes interact with the catalog manager which implement the catalog functions;

2. the use of Cronus data storage resources to implement the catalog data base; and

3. the distribution of the catalog data base among Cronus hosts;

8.4.2. Cronus Catalog Managers

There is a catalog manager process at each host that maintains part of the catalog. It is the object manager for objects of types CT_Directory, CT_Symbolic_Link, and CT_External_Linkage.

The catalog managers communicate with client processes by means of the standard Cronus IPC facility. Since the catalog hierarchy is distributed among Cronus hosts, different managers will have direct access to different parts of the catalog. Some catalog operations can be accomplished by a single catalog manager and some require the cooperation of two or more catalog managers.

For example, the *Remove* (directory) operation would normally be sent to the manager for the specified directory, and only that manager is required. The *Lookup* operation may require catalog managers on two hosts if the manager to which it is sent does not contain the subtree required to interpret the entire symbolic name.

A client process will not, in general, know which catalog manager is the best one to perform a given operation. For this reason, a client can initiate a catalog operation with any catalog manager. If the manager selected can perform the operation requested by itself, it will. If not, it will interact with other managers as necessary to perform the operation.

8.4.3. Implementation of the Catalog Hierarchy

Directories are stored in an object database. The catalog manager maintains a UID table for the objects it manages. Since the principal objects implemented by the catalog manager are directories, this table is called the Directory UID Table. The Directory UID Table maps the UIDs for directories into their object descriptors.

A directory contains zero or more catalog entries. The catalog entry for a (inferior) directory contains the UID of that directory. To access a directory given its UID, the catalog manager uses the Directory UID Table to obtain the object descriptor for the directory.

8.4.4. Distribution of the Catalog

8.4.4.1. Principles Affecting Distribution

Among the considerations influencing catalog distribution are:

1. The catalog should not be stored at only one site.

   This is a reliability consideration. The catalog should be distributed, and it should probably be replicated in some fashion.

2. The entire catalog should be distributed across several sites.

   This is a scalability consideration.

3. It should be possible to access the catalog entires for an object when the site that stores the object is accessible.

   This is a reliability consideration. Access to objects through the UID name space has this property since the information required to access an object, given its UID, is maintained by object managers. Access to objects through the symbolic name space should also exhibit it.

There are some further issues to consider associated with (2) and (4), and we discuss them in more detail in the next two subsections. The discussion includes elements of the implementation of the reliable system as well as the primal system, because these may impose constraints on the primal system design.

8.4.4.2. Dispersal Of The Catalog

This section examines the requirement that the catalog not be stored at a single site. The line of reasoning followed is essentially that that lead to the design of the Elan hierarchy [BBN 3796].

Directories are the basic unit of distribution for the Cronus catalog. Directories are implemented by Cronus as objects in an object database. The lookup operation follows the components of a symbolic name through a number of different directories, one for each component in the name (assuming it does not encounter a symbolic link). Unless there is a further restriction on the dispersal of the catalog, each directory could be at a different site from the previous one.

It is desirable to limit the number of sites that must be visited in a lookup operation. Two useful restrictions are to:

1. Require that the catalog structure for entire subtrees below a certain cut (the "dispersal cut") through the catalog tree be stored within a single site. We call a subtree that is rooted at the dispersal cut a "dispersal subtree".

2. Require that the catalog structure above the dispersal cut be stored within a single site. We call the structure above the dispersal cut the "root portion" of the hierarchy.

Restriction 1 ensures that lookup operations within a subtree that is below the dispersal cut can be confined to a single site. Restriction 2 ensures that the task of determining the site that stores a particular dispersal subtree can be confined to the site that stores the root portion of the hierarchy. As a result, lookup operations require at most two catalog sites.

It is useful to add a third property to the dispersal of the catalog:

3. The root portion of the catalog hierarchy should be replicated. Furthermore, a good way to replicate it is to maintain it at each site that maintains a part of the catalog (i.e. a dispersal subtree). The reasons for doing this are:

   - To distribute the load resulting from lookup operations among several sites.

   - To allow some lookup operations to be confined to a single site.

- To increase the availability of the root portion of the hierarchy.

Figure 8.3 illustrates how a simple name hierarchy might be dispersed among several hosts according to these three restrictions.

8.4.4.3. Replication of Catalog Information

The primary consideration for replicating catalog information is one of reliability. The objective is to ensure that Cronus objects with symbolic names are accessible symbolically whenever the sites that manage the objects are. This can be be insured by either mainting the catalog only on reliable service hosts or providing some dynamic replication in the catalog. To provide the most generality some capabilities should be present in the catalog managers to achieve the latter.

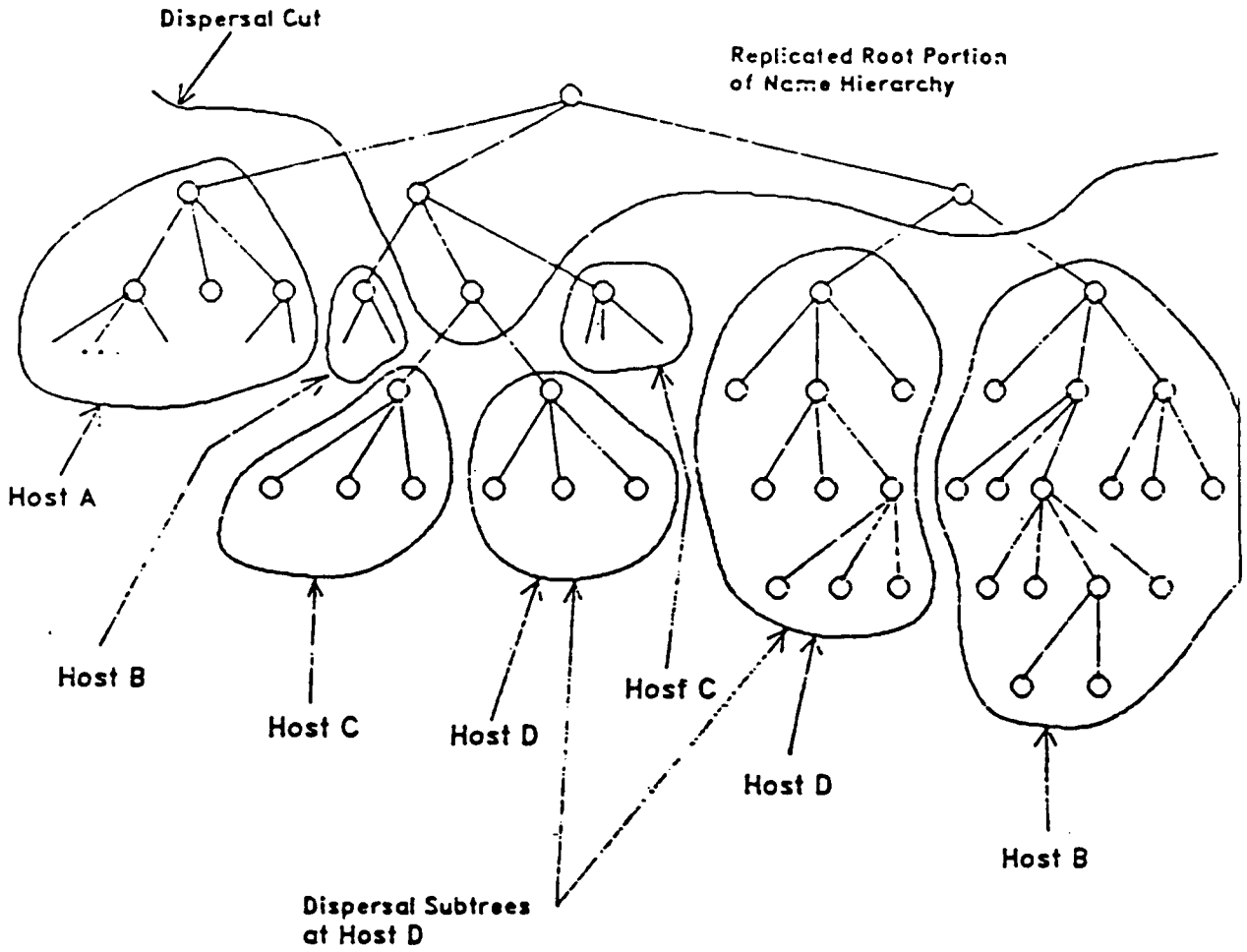
The problem of generalized replication in the catalog is similar to that of replicating many other Cronus object types. From this perspective, full replication below the dispersal cut is a matter of replicating the approprate directory objects starting with the root. Replication of critical directories can increase as necessary the availability of objects. This strategy reduces the catalog problem to an administrative one of deciding which directories need to be replicated, how many duplicates should be maintained and where each duplicate should be placed.

To control the replication of each individual directory entry would place an unnecessary burden on the implementation since the overhead associated with maintaining site lists and other information for each entry would be costly. Therefore, replication of the Cronus catalog is controlled at the directory level--each directory may be replicated or not, and the list of sites where copies of the directory are placed may be selected and modified. All copies are equivalent, none is considered primary, the manager receiving a *CreateEntry* or *RemoveEntry* locks all copies of the directory, makes the change locally and instructs managers for each of the c pies to make the same change and then release the lock.

Lookup operations may be p formed by a manager reponsible for any copy of the directory. The standard Cronus locate mechanis handle the location of a suitable site since the lookup operation is always invoked on a directory, id atified by its UID. The manager will attempt to resolve the pathname as far as possible, then pass the quest to a manager responsible for a copy of the root of the unmatc ed pathname component. This ob iously means that replicating each member of common pathname components at the same sites w l yield faster performance, but this is not required.

8.4.4.3.1. Synchronization Among Catalog Managers

The catalog managers must synchronize among themselves whenever an entry in a replicated directory is created or removed, and whenever a host which has been temporarily inaccessible is being reintegrated into the cluster. As with many other Cronus functions, automation of catalog replication is implemented through cooperation among the managers for the object type. For efficiency, we implement replication directly in the catalog managers, rather than building the catalog manager on a reliable



Dispersal of the Catalog
Figure 8.3

storage mechanism such as replicated files. While the approach we discuss applies to the Cronus catalog, it is also intended to be used as a base for more general replication services that might be applied to other Cronus components in the future.

Clearly, some form of concurrency control is needed to prevent conflicts and inconsistencies. Because changes to directories occur infrequently, we can prevent conflicts (simultaneous changes to the same entry), with little performance cost, by locking the copies of the affected directory while any change is being made, so that only one change can occur at any time.

We define the following basic operation replication control operations:

- Replicate existing directory
- Dereplicate existing directory
- Modify existing directory (add/delete/modify entry)
- Reintegrate host

In order to simplify the design, we will restrict ourselves to these functions. Other variants, such as create a new replicated directory, can be implemented from these and the existing catalog operations in the obvious manner.

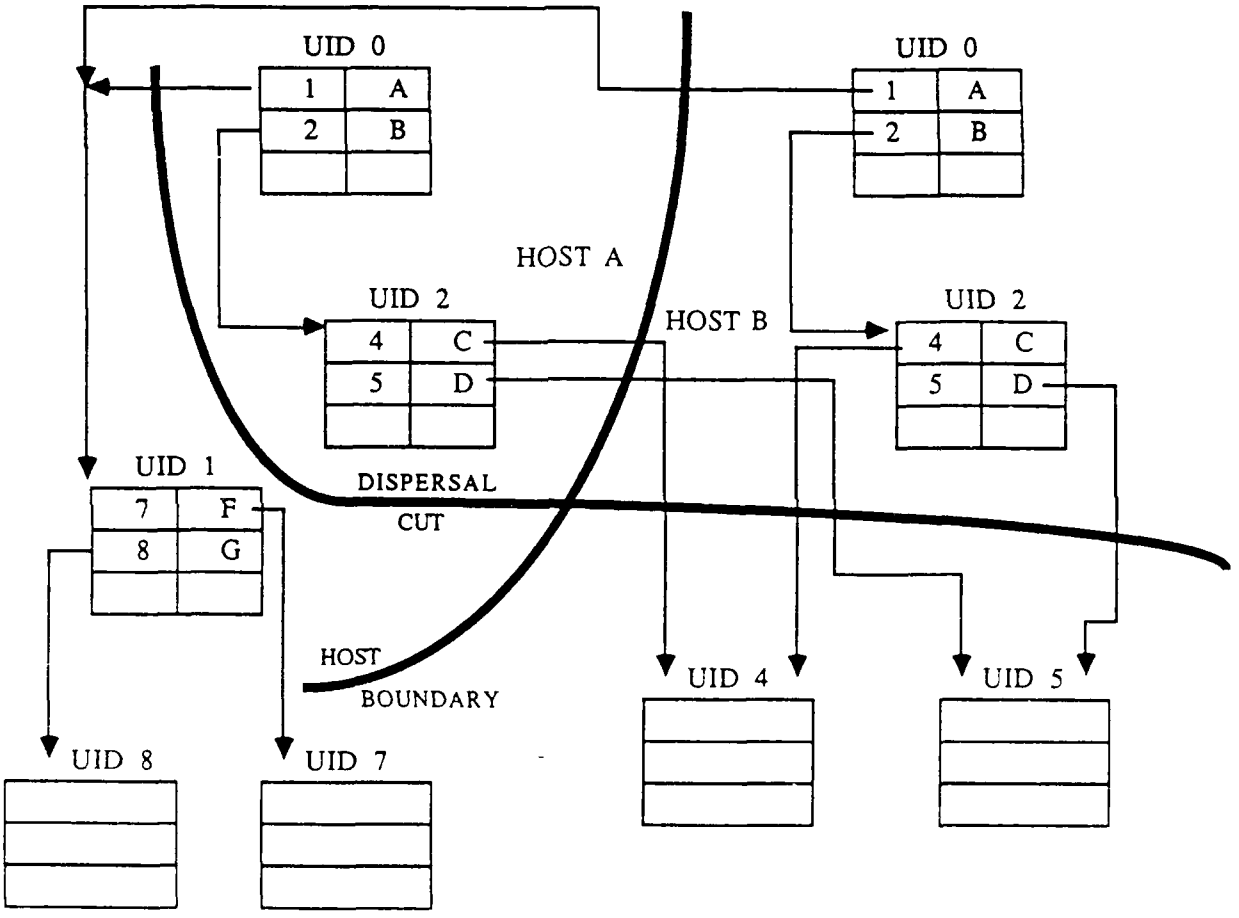
Our approach to maintaining consistency in the replicated portion of the hierarchy will be to lock the copies of a directory before modification and have the manager for the directory at one of the sites coordinate the changes to all copies, including unlocking the copies after the change has been made. We will discuss the management of updates in more detail later, when we discuss reintegration.

In Figure 4 we see a detailed representation of the replication of the root portion of the catalog hierarchy on two hosts, A and B. Note that the directories above the dispersal cut are truly replicated, having the same directory UIDs. The reader should remember that the contents of the replicated directories are also replicated (e.g they have the same entries), and that they have location independent semantics. That is, the entries consist of a symbolic name that is known globally (through the catalog) and a UID that is known globally (through the operation switch). With this background, we can now go on to discuss the operations in more detail.

8.4.4.3.2. Replicate

The *replicate* function takes a specified non-replicated directory and replicates it at specified host sites. That is, a copy of the directory, with the same UID as the original, and all the entries of the original will be created by the Catalog Manager on each site specified. To ensure consistency, existing copies of the directory are locked during the update. Thus, only after the new directory is allocated and its entries are complete is it made visible. Each copy of the directory includes a list identifying the sites where copies reside. The operation is coordinated by the Catalog Manager of the directory which receives the client's replicate request; this manager communicates directly with the Catalog Managers at the affected sites to complete the operation.

Replication in the Cronus Catalog



Replication in the Cronus Catalog
Figure 8-4

One issue raised by this method is whether the remote replications should be managed synchronously (waiting for remote operation to complete) or asynchronously (telling the remote Catalog Manager to start the operation and not waiting for completion). If the operation is synchronous, there are obvious performance implications for completion depending on how long the operation will take. For a large configuration this could be a problem. A time-out will be required for those hosts that are down or cannot respond. Asynchronous management means that it is hard for the originator to know when and if the operation was completed. It puts more of a burden on the reintegration procedure for making sure the operation is carried out successfully. One possibility in the asynchronous case is for the target to acknowledge start of the operation and not have the originator wait for completion.

The issue here is the definition of when an operation is complete. Strictly, an operation is complete only when all sites maintaining copies have successfully completed an update. However, it may be sufficient to consider an operation "complete" from the point of view of the initiator when it has been successfully accepted by a catalog manager and the manager responsible for each copy has been locked. Since the reintegration procedure will eventually cause the operation to be completed at all sites, relying on it to make sure the operation is completed at all sites appears adequate. Thus, the initiator's responsibility is to lock all sites with copies, start the operation at all sites with copies, and complete the operation on the local host. Once the operation is successfully initiated and updated locally, we assume that it will be completed on all hosts eventually, either as a result of the operation, or as a result of the reintegration procedure if any of the sites crash before the operation is complete.

The only problem with this approach is if a site *cannot* complete the operation due to problems such as lack of resources (e.g., no space to add new directories, etc.) In this rare case, the solution is to notify the operator of the resulting inconsistency through event logging of the monitoring and control system so that the problem can be manually corrected. The reintegration procedure can still be used in these cases to complete the operation at a later time, but presumably operator intervention will be required in some instances to correct the cause of the problem.

8.4.4.3.3. Dereplicate

The *dereplicate* function take a specified replicated directory and removes copies from identified sites. The algorithm is similar to plicate: first it locks the directory copies at each site, then it removes the copy from the identified site. moves the identified site from each site list, and unlocks the remaining copies.

8.4.4.3.4. Modify

The *modify* replicated directory operations (add, delete, change) also proceed along the lines of replicate/dereplicate, locking all copies of the directory, notifying all the remote Catalog Managers to perform the operation, and unlocking the directories.

8.4.4.3.5. Update

When a catalog manager returns to service after a temporary outage it scans the list of directories for which it is responsible. For any that are replicated, the manager retrieves an up-to-date copy by contacting an available site responsible for a copy of the directory.

Since our cluster is limited to a single local area network, we have not yet had to address the problem of reintegrating catalogs after temporary partitioning. When partitioning occurs, independent changes might occur to copies of a directory, with the result being that neither is clearly newer than another. Strategies such as version vectors, applied to individual directory entries could be used to resolve such conflicts in a future version of the catalog manager.

8.4.4.3.6. Administering the Dispersal Cut

User commands have been written which control replication of directories. A user may replicate or dereplicate a directory by specifying the host where the new copy will be placed or from which a copy should be removed. Another user command supports migrating directories from one host to another by replicating the directory to the new host and then removing the copy from its original location. Use of these commands is regulated by Cronus access control to the replicate and dereplicate directory operations. As with the directory operations they invoke, these commands may be applied to any directory, regardless of where it appears in the hierarchy.

Earlier, we referred to two other functions which are important in the practical administration of the replicated root portion of the Catalog Hierarchy. The first, move dispersal cut, can be thought of as a compound replicate/dereplicate operation whose semantics are: given a directory in the hierarchy move the dispersal cut to include it in the replicated portion by doing the appropriate replicate or dereplicate operations on the intervening directories. Conceptually this can be thought of as traversing the hierarchy and performing the individual replicate or dereplicate operations. Operationally, this function may be quite dangerous, so access control is used to limit its use to system operators.

The other function places a copy of the dispersal cut on a new host which will support cataloging functions. In this case one of the Catalog Managers walks down the root portion of the hierarchy and sends copies of each replicated directory to the new host. Since this is presumably done infrequently and at a time before the new host is supporting users, performance and synchronization issues are not issues.

8.5. COS Directories

8.5.1. Characteristics

Many resources and functions of a host continue to be used directly after the host has been integrated into a Cronus cluster. Also, many administrative tasks must be performed directly on the host. For example, directories where sources for constituent system commands are maintained usually exist on many machines in the cluster; users maintain directories and files containing mailboxes, sources, documents and other personal information; user accounts and access rights must be maintained for users

who may log directly onto a particular host. One goal of Cronus is to provide remote access to these resources, both to allow users to make use of Cronus development tools when manipulating these datasets from any access point, and to allow users to integrate information from a variety hosts which otherwise might require using cumbersome data transfer utilities. We also wish to support centralized administration for hosts in a cluster.

Most of the information that is maintained directly by users and administrators is stored in the file system by the Constituent Operating System (**COS**) running on the host. The data stored in each file system can be integrated into the Cronus file system through COS directories and COS files. These object types provide UID's which are mapped to actual native system directories and files; operations supported for Cronus directories and files are mapped by managers for COS directories and files into native operating system calls that create, read, modify and remove the native directory or file, as appropriate. The subdirectories of a COS directory, and all files contained in the directory and its subdirectories are automatically available by name. For example, consider that we create a COS directory for */usr/cronus* on the host *clxx*; this directory has subdirectories *source* and *bin*. If we catalog the UID for the COS directory as *:cronus*, we will be able to access the subdirectories by the names *:cronus:source* and *:cronus:bin*. A file called *client.c* in */usr/cronus/source* can be referenced by the name *:cronus:source:client.c*.

Two steps are required to attach a COS directory and its subtree to the Cronus catalog. The client first invokes the COS directory *create* request, supplying the COS pathname of the desired directory and directing the request to the host where the directory resides. The create request returns a Cronus UID which the client should record in a Cronus catalog *external link* entry. The external link entry was described in an earlier section, it allows the catalog manager to resolve the Cronus portion of the pathname (in our example, the *:cronus:source* component) and then forward the remaining portion to the manager for the COS directory that the external link references (*source*, *bin* and *source:client.c* in our example). By using the *lookup* request, its variants, and status requests, programs such as *list* can display the contents of COS directories just as they display the contents of Cronus catalog directories.

Currently, access to operations for creating and accessing COS directories and files are mediated by the Cronus access control mechanisms. The policy that this approach provides limits creation of COS file bindings to a selected administrative group for each host. We will soon improve the underlying mechanism to enhance this policy, allowing Cronus users to administer bindings to directories they own on constituent hosts.

One inevitable difference between conventional Cronus directories and COS directories arrises because COS directories can be manipulated through the constituent operating system without notice to COS directory manager responsible for them. In particular, COS directories may be deleted or removed without deleting or modifying the associated UID binding kept by the manager. Currently, the COS directory manager detects when a directory has been deleted, deletes the associated binding and notifies the client that the directory no longer exists. If the contents of the directory have been modified, those changes will be be reflected in the results of operations invoked through Cronus. In the future, we may encounter hosts where changes to the file cannot be detected in a timely fashion, and other strategies or administrative guidelines may be necessary.

9. Cronus File System

9.1. File System Overview

Cronus supports a number of different kinds of files, including:

- Primal files: The primal file is the most basic kind of Cronus file. Other kinds of Cronus files are implemented from primal files. A primal file is stored entirely within a single host, and is bound to the host.
- Reliable files: A reliable file is implemented by one or more primal files. Each primal file used to implement a reliable file contains all of the file data. The reliability of these files derives from the fact that the file is accessible as long as at least one of the primal files that implement it is.
- COS files: The COS file represents a file which is already provided and maintained on a particular host by its constituent operating system. The COS file manager allows such host files to accessed through Cronus, allowing them to be updated and maintained from remote locations.

The initial Cronus implementation (the "primal system") supports only primal files, which are implemented upon underlying single-host file systems. The next major Cronus release (the "reliable system") will support reliable files. Later system releases may support dispersed files.

This section also describes a single host file system, called the Elementary File System, which will be developed for each Cronus file host to serve as a common base of implementation support for Cronus file managers.

Primal files are Cronus objects. They have unique identifiers (UIDs), and may be given symbolic names. There is a Cronus object type CT_Primal_File.

9.2. Cronus Primal Files

9.2.1. Characteristics

Primal files cannot be moved from one host to another; the primal file system is partitioned among hosts that store primal files. The HostNumber component of the UID for a primal file always specifies the host on which the file is stored. A copy of a primal file can be created on another host, and the original can be deleted. The copy is a different primal file with a different UID; it just happens to contain the same data as the original file.

Like other Cronus objects, primal files are accessible to processes by means of the interprocess communication and operation switch (Section 6). There is a Primal File Manager process on each host that stores part of the primal file system. A client process accesses a primal file by invoking an operation on the file, in which the UID for the file and the operation to be performed on the file are specified.

The Primal File Manager that maintains a primal file also defines a mapping between the UID for the primal file and the information required to manage the file. The collection of information necessary to manage a primal file is called its descriptor. The file descriptor includes:

- UID of the creator;
- Date and time of creation;
- Date and time of last write;
- Access control list (ACL) for the file;
- Information necessary to find the file data on the storage media;
- Current size of the file;
- Other information (to be specified as needed)

Most of the operations provided by conventional file systems (create, read, write, etc.) are implemented for Cronus primal files. The design is discussed in terms of the normal life cycle of a primal file which includes:

1. The file is created.
2. Data in the file may be read or written by a client.
3. Information in the file descriptor may be read or written by a client.
4. The right to access the file may be granted to or revoked from other users.
5. The file may be deleted.

File creation involves: the generation of a UID; the creation and initialization of a descriptor for the file; and the binding of the UID and the file descriptor in the Primal File UID Table. Until data is written into the file, the file is empty. When a primal file is created by a Primal File Manager, it is created on that manager's host.

There is an issue regarding whether it should be necessary to open a primal file before reading or writing file data. One reason for "open" and "close" is to provide for reader-writer synchronization; another is optimization of read and write operations. The disadvantage is that open and close add complexity to the Primal File Manager because it must maintain state information for open files and deal with the problem of files opened which are never explicitly closed (e.g., because the client has crashed). Furthermore, if we require open and close, additional operations must be invoked on the file even when the read or write is for a small amount of data.

The Primal File Manager supports access to files without open and provides an open/close facility for clients that need it. A read or write without open is called a "free read" or a "free write". The client may then choose whether the additional overhead of opening and closing the file is worthwhile. For example, if we wish to write a simple log message when a process is initiated, we would probably choose the free write. If, on the other hand, we were copying a file, we would probably choose to open the files, incurring the overhead of initiation once, and gaining further system support for synchronization and data

integrity. A client process may read or write data in a primal file (subject to authorization considerations) without opening it, unless another process has opened the file in such a way that free reads and writes are forbidden.

Free reads and writes are synchronized in the sense that multiple reads and writes are serializable. This means that the File Manager will, in effect, perform each read or write operation in its entirety before performing another operation.

When a file is opened, two parameters specify the access state requested. One specifies either Read or ReadWrite access. The second specifies the type of reader-writer synchronization desired. There are two types of synchronization supported: "frozen" which permits either N readers or a single writer; and "thawed" which permits any number of simultaneous writers and readers. When a file is opened with "thawed" access, readers of the file see updates made by writers of the file. Opening a file with "thawed" access prevents other processes from opening it "frozen".

Thus, the access states defined for a file are:

free;
frozen read open;
frozen readwrite open;
thawed open;
(free) read in progress;
(free) write in progress.

A file may be opened so long as the access state requested does not conflict with the current access state of the file. Table 6.1 defines the compatibility of the access states with one another, and with read and write operations invoked by a client without previously opening the file. An OK for an (OPERATION, ACCESS STATE) entry in the table means that a client process can perform the operation on a file when the file is in the corresponding access state; a NO entry means that the operation will fail when the file is in the corresponding state; a DELAY operation means that the operation will be delayed until the operation in progress (and any others that may be queued) are completed.

The data in a primal file is a sequence of octets, numbered from 0 to N. The read operation specifies the first octet to be read and the number of octets to be read. The write operation specifies the octet position of the first octet to be written and N octets of data to be written.

In order to support file system recovery, data that is written to a file that has been opened for (ReadWrite, Frozen) access does not become part of the permanent file data until the file is closed. It is possible to close a file opened for (ReadWrite, Frozen) access in a way that aborts writes made to the file while it was open.

A file is open to a process. The Primal File Manager provides an operation which returns a list of the UIDs for the processes, if any, that have a given file open. Another operation returns a list of the UIDs for the files, if any, that a given process has open.

OPERATION	ACCESS STATE free	frozen read	frozen readwrite	thawed	read in progress	write in progress
frozen read open	OK	OK	NO	NO	OK	DELAY
frozen readwrite open	OK	NO	NO	NO	DELAY	DELAY
thawed open	OK	NO	NO	OK	DELAY	DELAY
free read	OK	OK	NO	OK	OK	DELAY
free write	OK	NO	NO	OK	DELAY	DELAY

Access State Compatibility
Table 9.1

When a process is destroyed with files open, the files are closed and any writes to (ReadWrite, Frozen) open files are aborted. The normal close operation may only be invoked by the process that opened the file. An alternate close operation can be used by other processes to close a file during cleanup.

A client can read the descriptor of a primal file. Some of the information in the file descriptor is changed as a side effect of operations on the file. For example, when a file is written, the date and time of last write is changed. There is other information that the client may wish to change explicitly.

Access to a primal file is controlled by its access control list (ACL). Access to a primal file may be granted to other users by adding entries to the ACL. Similarly, access to a file may be revoked from a user by removing the corresponding entry from the ACL.

Some file system support the notion of Delete, UnDelete and Expunge operations. The current design for the primal file system assumes that only Delete (called Remove) will be supported, but it is relatively straightforward to modify the specification of Cronus primal files to accommodate a Delete, Undelete, and Expunge model of file removal.

9.2.2. Crash Recovery Properties

If a primal file operation is invoked, the Primal File Manager normally acknowledges the operation, indicating the disposition of the operation (e.g., success, failure, and reason) and, depending upon the operation, to return any data requested.

The Primal File Manager does not acknowledge write requests until the data has been written to non-volatile storage. A client process can be sure that the data has been written when the acknowledgement is received, even if the Primal File Manager or its host should crash shortly afterward.

Primal File write operations are atomic with respect to host crashes. That is, if the Primal File Manager host should crash during a write operation, after the host and Primal File Manager have been restarted and the Primal File Manager has performed its recovery procedures, the write operation will have either occurred in its entirety or no part of it will have occurred. If the crash occurs after the data has been safely written but before the acknowledgement has been sent, the acknowledgement will never be generated.

This atomicity property is true for the Close-and-RetainWrites operation. That is, either none or all of the writes made while the file was open will have been performed.

9.2.3. Operations for Objects of Type Primal File

In addition to the generic operations the following operations are supported for primal files:

Open
Close
Sync
Read
Write
Truncate
Append
FilesOpenBy
OpenStatusOf
CloseProcessOpenFile
CloseAllProcessOpenFiles

The *Open* and *Close* operations provide an atomic transaction capability for a single primal file. At some later point, we may define explicit BeginTransaction, EndTransaction, and AddToTransaction operations which could be used to provide a capability for transactions that involve more than a single primal file.

In response to a Status operation, the Primal File Manager returns information about the status of the primal files it manages, such as the amount of free space, the amount of space used by existing files, the number of files it manages, the number of files currently opened, etc. This information will be useful to system operations personnel as well as to clients who might use it when deciding where to create primal files.

9.3. Reliable Files

9.3.1. Objectives

The principal motivation within Cronus for maintaining multiple copies of a file derives from reliability considerations. The objective is to increase the probability that the file will be available for access at any given time by keeping copies (in Cronus we shall call them images) of the file at a number of hosts. Although any given host that stores the file may fail, so long as at least one of the hosts maintaining an image is accessible, the file will be also.

Secondary benefits include performance improvements that may result from distributing the load due to file access among the hosts that store the file and from the possibility that client access to an image of the file maintained on its own host will be more responsive than access to an image on a remote host.

Increased file availability does not come for free. The cost is increased complexity in managing the files. Most of the complexity is a consequence of the fact that Cronus works to ensure the mutual consistency of the file images; when one image of the file changes, all others should be updated to reflect the change.

Furthermore, in the Cronus environment it is desirable to support concurrent access to files. For example, Cronus supports a form of multiple readers / single writer concurrency control for primal files. The same sort of concurrency control is provided for multi-image files.

Concurrency control requires that sites managing images of a file cooperate to synchronize client access to the file. There is complexity and overhead associated with this cooperation. In addition, since strong concurrency control mechanisms require the participation of more than one site, situations may arise where an insufficient number of file image sites are accessible to perform the concurrency control. Unless the system is willing to permit unsynchronized access to an accessible file image in such situations, some of the reliability benefits of multi-image files will be lost. The danger of unsynchronized access is, of course, that accessors may cause different images of a file to become inconsistent.

The Cronus approach to concurrency control for reliable files is based on the presumption that file availability is important enough that it is permissible to risk the consistency of file images and to grant access to file data when synchronization cannot be achieved. That is, when a choice must be made, file availability or survivability is considered more important than mutual consistency of file images.

The approach to concurrency control is to try to achieve strong synchronization prior to file access in order to maintain the consistency of the file images. However, should the synchronization fail because the file sites required to achieve it are inaccessible, the client will be informed and access to the file will be permitted only if the client gives explicit consent to continue.

This relaxed approach to concurrency control will be practical only if:

a. File access patterns are such that it is relatively unusual for multiple concurrent updates to occur.

b. Hosts are reasonably reliable so that host failures that prevent strong synchonization

are relatively rare.

c. There is a simple and inexpensive way to detect inconsistent images of a file. We believe that the Version Vector mechanism developed at UCLA [Parker 1983] is a good one for this purpose.

Experience with Cronus may show that there are some applications which require more absolute synchronization than this approach supports. If that proves to be the case, the support for reliable files will be augmented to include a file type for which more positive synchronization is supported.

9.3.2. Reliable Files as Composite Objects

A reliable file is a Cronus object of type, CT_Reliable_File. A Cronus Reliable File (RF) is a collection of one or more primal files, each of which represents an image of the reliable file. No two images of a reliable file are stored at the same site.

The number of images of a reliable file may change over the lifetime of the file, as may the sites which maintain the individual images. The desired number of images is called the cardinality of the file. The actual number of file images may be different than the file cardinality. For example, when a file is first created its cardinality will be greater than the number of images until all of the images are created. Similarly, if the cardinality of a file is changed, it takes finite amount time for the number of images to be adjusted. Thus, the cardinality is properly thought of as an objective.

A reliable file of cardinality = 1 is a migratory file. Although it has only a single image like a primal file, unlike a primal file it may be moved from one host to another.

Each Reliable File Manager (RFM) maintains a UID table for the reliable files that it manages. Unlike simpler objects, such as primal files, the management of reliable files requires the cooperation of RFMs. Each RFM participates in the management of a collection of reliable files (the ones in its UID table), but not all RFMs participate in the management of all reliable files.

Depending on the cardinality of a particular reliable file, a RFM may need to cooperate with 0 (cardinality = 1), 1 (cardinality = 2), or more (cardinality > 2) other RFMs. For each reliable file it manages, a RFM is directly responsible for carrying out the operations on a particular primal file that represents an image of the file. We shall sometimes refer to that image as the manager's image or as the local (to the manager) image.

When a client invokes an operation on a file, the underlying interprocess communication facility routes the operation to an RFM capable of performing it. Any interactions among RFMs that are required to perform the operation are transparent to the client process.

Access to the primal files that comprise a reliable files is limited to RFMs. No other process may directly access a primal file used to implement a reliable file, even if the process has the UID for the primal file; this is enforced by the Cronus access control mechanism.

For Cronus, RFMs reside only on sites that also have primal files managers (PFMs). The manager's image of the file is stored at the manager's site. RFMs, of course, access the file images through PFMs in the normal fashion.

There is an issue regarding the relation of RFMs to PFMs. They could be implemented either as two completely separate managers which communicate by means of interprocess communication or as a single, combined manager for both CT Primal File and CT Reliable File. The initial implementation of reliable files will be accomplished by means of RFMs that are separate from the PFMs. Later implementations may integrate the RFM functions into (some of) the PFMs.

In addition to the information maintained in descriptors for primal files, object descriptors for reliable files contain the following information:

File Cardinality;
ID of primary site (see below);
Version vector for the local image of the file
(see below);
Version vector for the local image of the
descriptor (see below);
List of UID's for the primal files that implement
images of the file.

9.3.3. Synchronization Considerations

In order to maintain the consistency of images of reliable files and the integrity of internal file data (for primal as well as reliable files), Cronus must control and synchronize the manner in which clients access the files.

The general Cronus approach to synchronization for reliable files can be characterized as a best effort approach consisting of the following steps:

1. try to synchronize access;
2. if synchronization cannot be achieved permit access if the client so desires;
3. be prepared to detect and deal with inconsistencies that may result from unsynchronized access later.

A specific concurrency control mechanism must be chosen. Although much has be written about concurrency control and synchronization for multiple copy files and data bases, there is little practical experience on which to base a choice. We have decided to use a simple mechanism for Cronus. Should the mechanism prove to be inadequate (for example, becau it cannot achieve synchronization often enough, given the failure patterns observed in Cronus), it will be replaced with a more capable (and complex) one.

Synchronization will be accomplished by means of a primary/secondary image approach. Each reliable file will have one primary image and one or more secondary images. All attempts to synchronize access to a reliable file will require synchronization with the primary image. We refer to the manager of the primary image as the primary manager for the file; managers of other images are called secondary managers.

When a client attempts to access file data in a way that requires synchronization, an attempt will be made to synchronize with the primary image of the file. If the client's access attempt is initiated with the manager for the primary image, synchronization occurs as for primal files. If the access attempt is initiated with the manager for a secondary image of the file, the secondary manager interacts with the primary manager to gain the appropriate kind of access (non-exclusive read, exclusive write).

RFMs use a locking discipline to support synchronization. This discipline works roughly as follows. When an attempt to open a file for reading is handled by a secondary manager, the manager tries to set its lock for the file to "reserved for reading". The attempt to set the lock fails if the file is already locked for writing. Next, the manager interacts with the primary manager to try to set the primary manager's lock for the file. If this succeeds, the secondary manager sets its lock to "locked for reading" and proceeds with the open. If the primary has the file locked for writing, the secondary manager clears its lock and reports to the client that the file is busy. When the file is closed, both the local lock and the primary manager's lock for the file are cleared. Attempts to open a file for writing are handled in an analogous fashion.

The reliable file system supports the notion of free reads and writes. For a free read the synchronization outlined in Table 9.2 is performed by the file manager which handles the client's read, but no attempt to synchronize with the primary manager is made. Free write operations require synchronization with the primary manager.

If sychronization for any operation fails because the primary manager cannot be reached, the operation may proceed, but only with the explicit consent of the client, and, of course, at some risk. The risk is that different images of the file may be undergoing unsynchronized access, and, as a result, the file images may diverge into inconsistent states.

A client may specify its intent with regard to unsynchronized access when it initiates a file operation by means of an optional operation parameter. Alternatively, the client may choose not to specify the action to be taken when it invokes the operation, in which case, if synchronization cannot be achieved, the manager will ask whether it should proceed with or abort the operation.

Inconsistent images of a file can be detected by means of the version vector mechanism developed at UCLA. A version vector for a reliable file, RF, is a set of N ordered pairs, where N is the number of sites at which RF is stored. A particular pair (Si, Vi) counts the number of times updates to RF were initiated at Si. Thus, each time an update to RF originates at Si, Vi is incremented by one. The version vector is part of the object descriptor for RF.

Two images of a reliable file are said to be consistent if the modification history of one is the same as or is an initial subsequence of that of the other. It can be shown that two images are consistent if one of the vectors is at least as large as the other in every (Si, Vi) pair. The larger vector is said to dominate the smaller, and the image corresponding to it represents a later, consistent version of the image corresponding to the smaller vector. If two vectors are such that neither dominates the other (that is,

some pairs in one are larger than some pairs in the other and vice versa), then the corresponding file images are inconsistent with one another.

Since the descriptor for a file may undergo modification independently of the file data, descriptors for reliable files also have version vectors.

The question of when version vectors for file images should be compared and what to do if they are not equal is discussed in Section 9.3.6.

9.3.4. Interactions Among Reliable File Managers

RFM's must interact with one another in order to maintain reliable files. For example, when a reliable file is updated, the new file data must be transmitted to each site that has an image of the file.

Occasionally a RFM that must participate in such an interaction will be inaccessible. It is important that when, if ever, such a RFM becomes accessible the interaction occur. It is the responsibility of the initiating RFM to ensure that the interaction occurs. The mechanism used by RFM's to do this is as follows:

Each RFM maintains a PendingActions data base which contains a record for each operation it was unable to completely perform due to its inability to interact with other RFM's. Each such record includes:

> the UID of the reliable file;
> a specification of the action required to complete
> the operation;
> a list of the sites at which the action must be
> performed (for some actions, this list may be empty).

Whenever the RFM is unable to complete an operation, it adds a record to the PendingActions data base to describe the actions necessary to complete the operation. Subsequently, at regular intervals, the RFM scans the PendingActions data base and for each record, it attempts to perform the necessary interactions. If the RFM succeeds in performing some, but not all, of the interactions, it updates the record. When all of the interactions described by a record are successfully performed the record is removed from the data base.

The actions that may be found in records in the PendingActions data base include:

a. Acquire sites to store images of a file.

b. Update the descriptor for a file.

c. Update a file itself.

When a RFM comes up for the first time, its PendingActions data base is empty, and if sites and the network never failed the data base would remain empty.

The PendingActions data base should be stored in a reasonably reliable fashion. It is probably adequate to store it as a primal file on the RFM's local site.

9.3.5. Operations on Reliable Files

The operations supported for primal files are also supported for reliable files. Three additional operations are supported for reliable files. The Change_Cardinality operation changes the cardinality of a reliable file. The File_Sites operation produces a list of the sites that are thought to be maintaining images of the file, with the primary file site distinguished. The Move_Image_To_Site operation moves a file image from one site to another (removing the image at the source site).

The design of reliable files is conveniently described in terms of the normal life cycle for a file, which is much the same as that for a primal file. The principal exception is that the cardinality of the file may change. The life cycle includes:

a. The file is created.

b. Data in the file may be read by a client.

c. Data in the file may be written by a client.

d. Information in the file descriptor may be read by a client.

e. Information in the file descriptor may be written by a client.

f. The cardinality of the file may be changed.

g. The file may be deleted.

The following sections discuss these operations.

9.3.5.1. Creating Reliable Files

A reliable file must be created before data can be written into it, and until data is written into the file, the file remains empty.

To create a reliable file, the client invokes the Create operation specifying the cardinality of the file as a parameter. The RFM that receives the Create operation becomes the primary manager for the file.

For the initial implementation of reliable files, clients may exercise control only over where primary file images are maintained. If the Create operation is requested by means of InvokeOnHost, then the RFM at that host becomes the primary manager; otherwise, the RFM selected by the interprocess communication facility becomes the primary manager. Later implementations may provide means for client processes (as well as for users through the user interface) to exercise control over the initial placement of secondary images. After images are in place, the Move_Image_To_Site operation can be used to move an image from one site to another.

When a RFM receives a Create operation, it:

a. Creates a (empty) primal file for the primary image of the reliable file, and obtains its UID (UID_pf).

b. Allocates a UID (UID_rf) for the reliable file, and makes an entry for it in its UID table;

c. Creates and initializes a descriptor for the reliable file. The following descriptor fields are initialized:

   The cardinality;
   The primary site;
   The file version vector and descriptor version vector;
   The list of UIDs for images is initialized to include UID_pf.

d. Returns UID_rf to the client, indicating that the Create succeeded.

Secondary images of the file are not created until the file is written the first time. (That is, after a free write or after the file is opened, written into and closed).

When a reliable file is first written and whenever the file cardinality is increased, the RFM selects sites to store images of the file. The acquisition of new sites involves three steps:

a. The selection of the new sites.

b. Obtaining commitments from the RFMs at the selected sites to store images of the file.

c. Updating file descriptors at each of the file sites to reflect the new sites.

The RFM acquisition procedure is structured so that an RFM need not, as part of a single acquisition attempt, acquire every site required to support a file's cardinality. An RFM can support operations on a reliable file even if not all of the desired images of the file have been created. When an RFM is unable to acquire all the sites necessary to achieve the desired file cardinality, it creates a record in its PendingActions data base to ensure that the additional sites will be acquired.

9.3.5.2. Reading Reliable Files

Reading a reliable file is similar to reading a primal file. File data may be read by means of a free read operation, or by opening the file prior to performing read operations. In either case the interprocess communication facility delivers the operations to an RFM that manages the file.

There are several differences in dealing with reliable files which are visible to a client. These include the following:

a. The interaction between the RFM that receives the operation and the primary RFM for the file in order to achieve synchronization is not visible to the client. However, should the synchronization fail because the primary RFM is inaccessible, the client will be informed and given an opportunity either to continue with the access or to abort it.

b. A client process can obtain a list of the sites that have images of a reliable file, and it can choose which RFM to deal with to access the file. For example, it might choose the primary RFM, or, if an RFM happens to reside on the host it does, it might choose that one.

c. After it opens a file, the client should continue to deal with the same RFM for operations on the open file until it closes the file.

9.3.5.3. Writing Reliable Files

Writing a reliable file is similar to writing a primal file. The principal differences are essentially those noted above for reading reliable files: the required synchronization may fail due to the inaccessibility of the primary manager for the file, in which case the client must decide whether to proceed at some risk or to abort the write; the client may choose the RFM with which it deals; and, after it has opened a reliable file for writing, a client should deal with the same RFM for operations on the open file until it closes the file.

File data must be updated after a free write or after a file opened for writing has been closed (if writes have actually been made and are to be retained).

The RFM at which the writes are performed is responsible for distributing updates to the other file images. It does this by interacting with the other RFMs sites in the following way:

a. It increments its (Site, Version) element of the file version vector.

b. It attempts to interact with each other RFM that manages an image of the file.

c. Should it fail to complete the image update with any RFM, it adds a record to the PendingActions data base specifying the file and the RFMs it was unable to update.

The actual update procedure for a particular image involves several exchanges between the initiating

RFM (iRFM) and the responding RFM (rRFM), and works roughly as follows:

a. iRFM does InvokeOnHost(SiteOf(rRFM), UID, UpdateImage, DVV, FVV);

where UID is the UID of the reliable file, DVV is the version vector for the file descriptor, and FVV is the version vector for the file itself.

b. rRFM compares both DVV and FVV against the descriptor and file version vectors it maintains for UID. Assuming that DVV and FVV dominate the corresponding version vectors at rRFM, rRFM returns to iRFM a SendTheDescriptor message. (Section 9.3.6 discusses what happens if iRFM's version vectors are dominated by or are incompatible with rRFM's.)

c. When iRFM receives the SendTheDescriptor message, it sends the new value of the file descriptor to rRFM in a HereIsTheDescriptor message.

d. rRFM receives the file descriptor and updates its copy of the descriptor. It then returns iRFM a SendTheFileUpdate message.

e. When iRFM receives the SendTheFileUpdate message, it transmits the file update to rRFM in a HereIsTheFileUpdate message. Depending on the nature of the changes to be made to the file image, the update may be transmitted by sending the entire file or by sending only the changes that need to be made to the file to update it.

f. Finally, after it has stored the new file data in the primal file that holds its image of the file, rRFM returns an UpdateImageSucceeded message to iRFM.

9.3.5.4. Other Operations

This section describes the Change Cardinality and Move Image To Site operations. Both operations require synchronization with the primary manager.

Change Cardinality is used to change the number of images the system tries to maintain for a reliable file. An increase to the cardinality is accomplished by execution of the acquisition procedure described in Section 9.3.5.1. Decreasing the cardinality is roughly the inverse of increasing it. The performing manager selects a site or a set of sites which currently maintain images of the file and asks the manager at each to agree to discard its image of the file, and to remove the file from its UID table. After each agrees, the performing manager instructs each to discard the image and the remaining managers to update their descriptors for the file.

Move_Image_To_Site moves a file image from one site to another, preserving the file cardinality. The parameters of the operation are the file UID, the site of the image to move, and a new site to hold the image. The operation involves creating an image of the file at the new site, discarding the image at the old site, and updating the descriptors held by all managers of the file to reflect the change.

9.3.6. Use of Version Vectors

Version vectors are used to detect inconsistent images of reliable files. In the current design, both the descriptor for a file and the file itself are protected by version vectors.

Version vectors are compared in two situations:

a. When an image of a file is updated. The RFM initiating the image update supplies its version vectors, and the responding RFM compares them with its own.

b. When an attempt is made to lock a file for read or write access. The secondary RFM attempting to lock the file supplies the primary RFM with its version vectors and the primary RFM does the comparison.

In each situation, both the descriptor version vector and the file data version vector are compared. There are four possible outcomes for the comparison of version vectors:

a. The supplied version vector is the same as the local version vector.

b. The supplied version vector dominates the local version vector.

c. The supplied version vector is dominated by the local version vector.

d. The two version vectors are incompatible.

The actions taken for these outcomes depend upon whether image updating or file locking is taking place.

For updating, the version vectors are compared by the RFM whose image is about to be updated. The various comparison outcomes and the actions to be taken for each are:

a. The supplied version vector is the same as the local version vector. Since the updating RFM increments its element of the version vector prior to sending it for comparison, if the RFMs are behaving properly, this case should not occur. If it does, some RFM has been misbehaving. The update should be deferred and the operations staff should be alerted by means of a message to the Monitoring and Control System.

b. The supplied version vector dominates the local version vector. This is the normal case, since the local image is being updated. In this case, the image update should proceed.

c. The supplied version vector is dominated by the local version vector. In this case, the local image is more recent than the one that is to replace it. The update should be aborted, and the local version should be used to update the remote version.

d. The version vectors are incompatible. This detects an inconsistency. The update should be deferred until human intervention can clear up the problem.

In the locking situation, the version vectors are being compared by the primary RFM for the file in question:

a. The supplied version vector is the same as the local version vector. This should be the normal case, and locking can proceed.

b. The supplied version vector dominates the local version vector. In this case, the primary image is obsolete, and should be brought up to date. If the file is being locked for writing, the locking should proceed, and the local image can be updated when the file is closed. If the file is being locked for reading, there are two possibilities. Either, the primary file image could be updated before proceeding with the locking, or the locking could proceed and the file could be updated when the lock is cleared.

c. The supplied version vector is dominated by the local version vector. The secondary image should be updated before proceeding. If the file is being locked for reading, then the file image at the secondary site should be updated so that the client is given access to the most current file data. If the file is being locked for writing, then the secondary file image must be updated first to avoid incompatibility.

d. The version vectors are incompatible. If the file is being locked for reading, the locking may proceed, but an attempt to signal a user or operator to resolve the incompatibility should be made. If the file is being locked for writing, the client should be informed of the incompatibility and given an opportunity to resolve it. The client may proceed without resolving the incompatibility, in which case the write is treated as an unsynchronized write.

9.4. COS Files

9.4.1. Characteristics

The motives for supporting COS files and directories were discussed in the COS directory description of the catalog section. Briefly, we wish to provide remote access to file resources to files and directories maintained by the constituent operating systems so that the information they contain can be manipulated and integrated by a user from any point in the cluster. This also allows many cluster host administrative activities to be moved to a common location.

Catalog entries for COS files are usually introduced into the Cronus catalog by creating a link to the COS directory that contains the files. However, individual COS files may also be created by supplying a COS pathname to the manager responsible for COS files on the intended host, and entering the UID returned by the create request into the Cronus catalog. Thereafter, clients may open the COS file by specifying its UID, retrieved from the COS directory or Cronus directory, as appropriate. Using the descriptor returned by the *open* request, normal Cronus file operators may be performed, namely *open*, *read*, *write*, and *close*. This allows Cronus file utilities, such as text editors, file copy utilities and application programs to be indifferent to whether their targets are Cronus files or COS files. This enables not only remote file editing or remote access to a mailbox, but also allows programmed, systematic update

of these files to be performed, as might be done by a software distribution program which periodically updates copies of programs and program sources at a collection of cluster sites.

Currently, access to operations on both COS directories and COS files are mediated by the Cronus access control mechanisms. This approach limits creation of COS file bindings to a selected administrative group for each host. We will soon improve the underlying mechanism to enhance this policy, allowing Cronus users to administer bindings to files they own on constituent hosts.

As with directories, COS files are inevitably different from Cronus files because COS files can be manipulated through the constituent operating system without notifying the COS file manager responsible for them. COS files may be deleted or removed without deleting or modifying the associated UID to file mapping kept by the manager. Currently, the COS file managers detects when a file has been deleted, deletes the associated binding and notifies the client that the file no longer exists. If the contents of the file have been modified, those changes will be generally be reflected in the results of operations invoked through Cronus. In the future, we may encounter hosts where changes to the file cannot be detected in a timely fashion, and other strategies or administrative guidelines may be necessary.

9.5. Elementary File System

9.5.1. Introduction

The Elementary File System (EFS) is an easily ported single host file system that serves as a common base of implementation support for Cronus file managers on Cronus Generic Computing Elements (GCEs) configured with disks, on the UNIX system, and on the VAX. The underlying implementation of the EFS is constituent host dependent, but the interface presented to the Cronus File Manger is uniform. As a result, portability of the File Manager is enhanced, and the cost of integration of new hosts is reduced. The EFS was originally developed as a primitive file storage capability for the GCE mass storage devices.

The two principal design objectives of the EFS are:

1. Sufficient functional capability to support the Cronus distributed file system.

2. Simplicity and efficiency.

   The principal users of the EFS will be object managers. Client processes will seldom, if ever, directly access files through the EFS. Therefore, only the most basic file operations need be supported. More complex file functions can be supported by the object managers themselves. Simple steps have been taken in the internal organization of the EFS to support effective crash recovery and system restart procedures.

The Elementary File System will have the following characteristics:

1. The name space for EFS files is flat. Names for EFS files are called FileIDs, and they are fixed length bit strings. FileIDs are not Cronus UIDs. A FileID is unique on the EFS that generated it, but it is not unique across all Cronus hosts. The EFS is a Cronus object in much

the same way that the existing UNIX or VMS file systems are Cronus objects, but

2. A EFS file is not a Cronus object.

3. File data is organized as a sequence of fixed length blocks. File i/o is sequential, and is block oriented. The basic file i/o operations are:

   ReadEFSFileBlock(FileID, BlockNumber, Buffer), and
   WriteEFSFileBlock(FileID, BlockNumber, Buffer).

4. There are no open or close operations. No setup is necessary to read data from or write data to an existing EFS file.

5. It is necessary to create a EFS file before writing data to it. This is accomplished by the

   CreateEFSFile()

   operation, which creates an empty EFS file and returns its FileID.

6. EFS files are deleted by the

   DeleteEFSFile(FileID)

   operation.

7. There is no access control for EFS files. Possession of the FileID for a EFS file is sufficient to access the file.

The EFS will normally be accessible only to Cronus Services. The primal file manager is an example of such a service. These services provide controlled access to the objects and operations that they implement, as described in Section 9.

In addition to supporting the local primal file manager, the EFS may be operated on as an object to permit remote access for maintenance and debugging purposes. There is a single access control list (ACL) associated with access to the entire EFS through the EFS File Manager. Only a very few principals will be on the ACL for a EFS. An example of a principal which might be granted access to the EFS is the "System Maintenance" principa

9.5.2. File Formats

The following description of the Elementary File System structure assumes that a disk can be represented by a series of fixed length blocks. In the Cronus ADM, the storage may be:

a disk drive on a GCE;

a disk device in a UNIX system; or

a contiguous file on the VAX/VMS.

The EFS makes few demands on the underlying recording medium, and it is relatively easy to see that most potential Constituent Operating Systems will provide a construct upon which the EFS can be built.

File disk blocks are self-identifying for reliability purposes. Each block includes a header that contains the FileID and the block number. The file header in each block contains a NextBlock pointer which is the disk address of the next block, if any, in the file. The NextBlock pointer in the last block contains a special value marking the end of file.

There is a FileID Table which provides a mapping between FileIDs and the disk address of block 0 of the file (see Figure 1). The FileID Table is as a file with a well-known FileID (FileID = 1). Its block 0 will be stored at a known disk address (with an alternate copy stored at another location to prevent loss of data in case the primary block is bad). The FileID Table is a hash table.

There is a FreeDiskBlock table which records the disk blocks that are available. The FreeDiskBlock table is a bit table stored in a file with a well-known FileID (FileID = 2). Its block 0 is stored at a known disk address. When a file is deleted, its blocks are recorded in the FreeDiskBlock table, and the FileID field in the headers of each of the blocks is cleared. As disk blocks are needed they are allocated using the FreeDiskBlock table.

There are two types of EFS files. The type of the file is contained in the header of block 0. The types of EFS files are (see Figure 2):

a. Short file.

This is a file, all of whose data will fit within block 0.

b. Normal file.

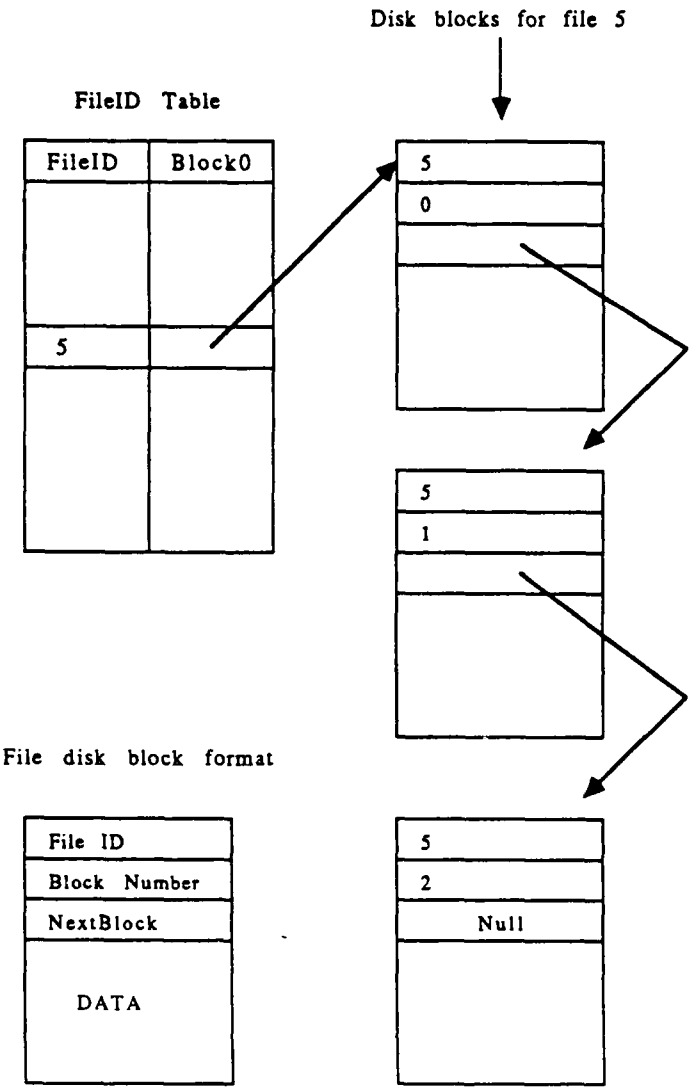
This is a file whose data will not fit within a single block.

A Normal file may contain index blocks which allow random access to the file. By convention, the first of these blocks is given block number -1, and contains:

i. A block index which holds the disk address of blocks 1 through N of the file; and
ii The disk addresses for two overflow blocks, named OverflowBlock1 and OverflowBlock2, which can be used to find the block index entries for blocks numbered greater than N.

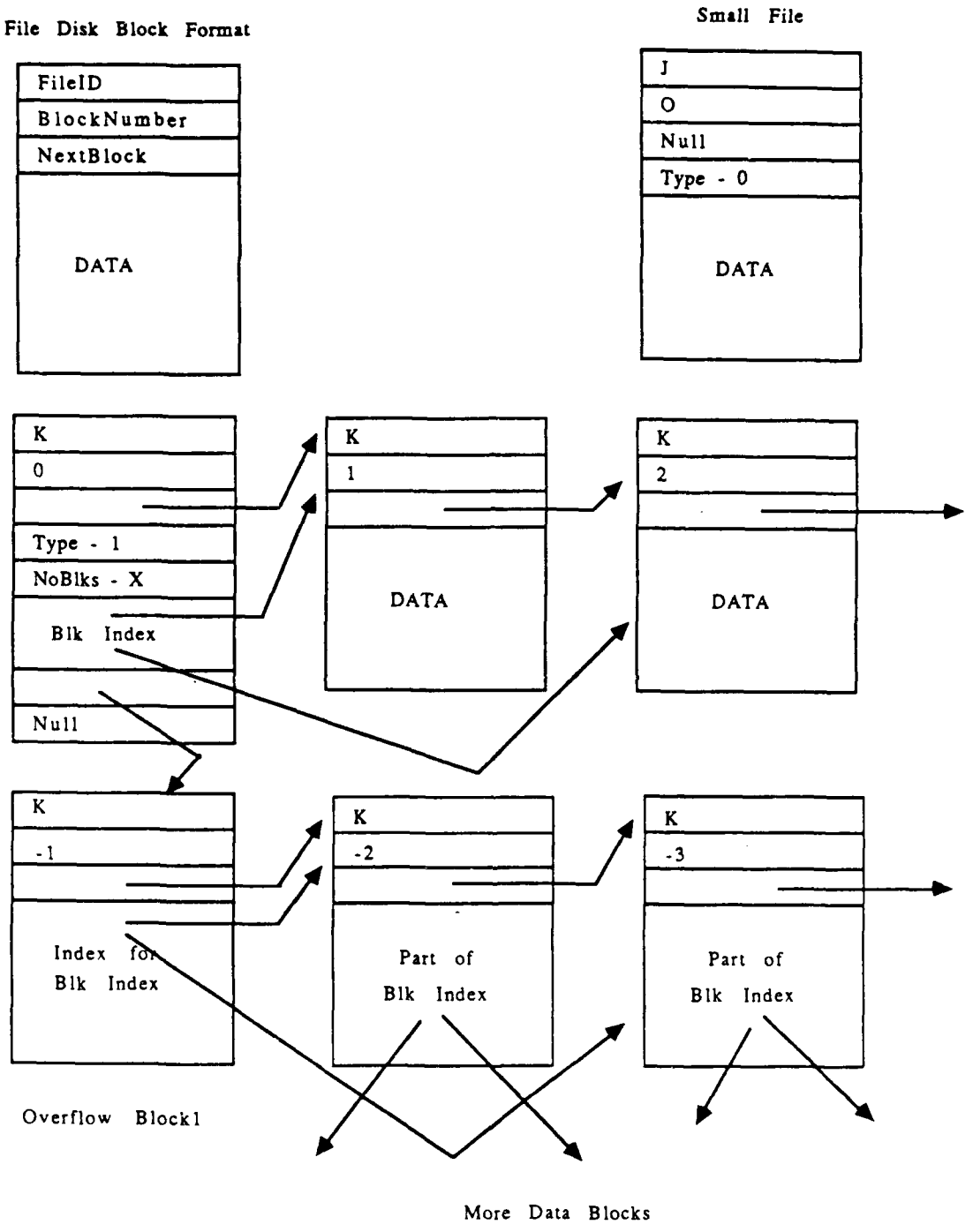
If the file is very large, not all of its index will fit into block -1.

OverflowBlock1 is used as an index for blocks which store part of the block index which will not fit in block -1. Specifically, if block -1 can store indices for blocks 1 through N, if OverflowBlock1 can store M disk addresses as indices, and if each block it indexes can store P disk addresses, OverflowBlock1 can provide access to indices for M*P additional blocks, numbered (N+1) through (N+M*P). The block



EFS File Table
Figure 9.1

Random Access GCE Files



EFS File Types
Figure 9.2

index for the Normal file shown in Figure 2 overflows block -1 into OverflowBlock1, and is small enough that it doesn't require OverflowBlock2.

OverflowBlock2 provides an additional level of indirection for very large files. It contains an index for blocks which are used in the same manner OverflowBlock1 is. If OverflowBlock2 can hold Q disk addresses as indices, then it can provide access to indices for M*P*Q blocks, numbered (N+M*P+1) through (N+M*P+1+M*P*Q).

By convention the BlockNumber for OverflowBlock1 is -2. Any index blocks referenced by OverflowBlock1, as well as OverflowBlock2 (if present), and any index blocks it references directly or indirectly are assigned BlockNumbers in a negative sequential fashion starting at -3 in the obvious manner.

Some constituent hosts will have multiple disks (in the case of UNIX, these may actually be disjoint regions on a single physical disk, and in the case of VMS, they would be multiple contiguous files). Part of the FileID specifies the disk on which the file resides. The CreateEFSFile operation takes an optional parameter which specifies a disk. If the parameter is supplied, block 0 and all subsequently created blocks of the file are allocated on the specified disk. If the parameter is not supplied, block 0 and subsequent blocks are allocated on the disk the EFS sees fit.

9.5.3. Disk Salvaging

There is a BadDiskBlock table which holds the disk addresses of bad disk blocks. The BadDiskBlock table is stored in a file with a well-known FileID (FileID = 3).

There is a EFS disk salvage operation which can reconstruct the FileID table, the FreeDiskBlock file, and the BadDiskBlock file, and reset the NextBlock pointers in files.

The salvager may encounter files with missing blocks. When it does, it will fill in any hole it encounters with a newly allocated filler block, linking the filler block into the file where the hole was. The FileID of the filler block will be set to the ID of the file, and its BlockNumber will be set to a special BlockNumber which identifies it as a filler block. The only data in a filler block will be the BlockNumbers of the previous and next file blocks which contain data. Higher level software can be used to recover the data in a file which contains filler blocks.

As the salvage procedure encounters bad disk blocks, it records them in the BadDiskBlock file. If it encounters a bad block which is part of a file, the salvager will remove the block from the file and substitute a newly allocated replacement block by linking it with the other blocks of the file in place of the bad block. The FileID of the replacement block will be set to the ID of the file, and its BlockNumber will be set to a special BlockNumber which identifies it is a replacement block. The only data in the replacement block will be the BlockNumber of the block it replaces. This will make it possible for higher level software to recover the data in other blocks of the file.

10. Input/Output

10.1. Introduction

Devices, such as line printers, tape drives, or terminals are integrated into the Cronus system as subtypes of generalized objects. These generalized objects serve to catagorize devices by the way in which requests for the device are submitted and manipulated. This strategy provides a richer organizational structure than the simple model of device independence offered by traditional single host operating systems such as Unix, where most devices are abstracted to appear similar to either a sequential or random access file. For example, the existing line printer driver is implemented as a subtype CT_File so that utilities which normally direct their output to files may be directed to a line printer object; the data written to the printer will be queued and printed in order. An alternative strategy for a line printer would be to view it as a queue, similar in operation to a directory; each entry would represent one queued request. The queue could be listed, entries removed or their order rearranged.

To date, for devices other than line printers, we have generally used the constituent systems to provide access to host peripherals. The remainder of this section presents some ideas on how devices might be organized around a stream object; we expect that as our experience with integrating devices into Cronus grows, many more strategies will be added to the list of approaches.

10.2. Operations on devices

Devices are objects of type CT_Device, which is a subtype of type CT_IOStream, and implements the standard operations of that type:

Open
Close[12]
IOLock
Read
Write
IOStreamsOpenBy
OpenStatusOf
CloseProcessOpenIOStreams
CloseAllProcessOpenIOStreams

In addition to these operations, device objects also implement a number of special-purpose operations, for example, a tape drive or a disk drive have a Seek operation to allow writing or reading to be done from a particular position in the medium which the device uses[13]. The details of individual device-object

[12]Open and close are used for synchronization. They are also used to trigger those actions that many device managers will wish to perform (e.g., hanging up a modem when the last process closes its output to the terminal, issuing a form-feed when a process opens the lineprinter) when the device gets accessed.
[13]Other special operations individual device managers are likely to implement are: density and format control for tape and disk drives; many devices may be turned off-line by software; printers will have page-length, page-width, and font controls, and so on.

operations will be specified as actual devices are added to the CRONUS cluster[14].

We anticipate a hierarchy of object types, breaking down into finer and finer distinctions. For example, CT_IOStream -> CT_Device -> CT_printer -> CT_lineprinter. Just as there are several kinds of I/O-stream objects, there may be many kinds of lineprinter object, perhaps one for each kind of lineprinter, or there may be page printers and graphics printers.

Device object managers also will commonly refuse a request for "frozen" access. In addition to the exclusivity of access provided by frozen access, one also gains the ability to cancel the writes which have been done to the object. This latter ability cannot be implemented on devices in any meaningful way, so this form of access is not allowed by the device's manager[15]. One may open devices for exclusive access, of course.

10.3. Implementation overview

For each device object on a host there is a manager for the device. Device managers may manage multiple devices (for example, a host might have only one line-printer manager for all of its lineprinters, or may have a single manager that manages both tape-drives and disk-drives[16]), or a manager may manage a single device. Which of these approaches is taken will depend entirely on the implementation, and is not within the scope of this document. When started, the device manager registers the UIDs for its devices with the operation switch on its host, so that the Cronus IPC mechanism delivers operations on the device object appropriately.

10.3.1. The use of large messages for device I/O

We expect that most I/O devices will be done using a stream interface as supported by Cronus large messages, in order to avoid passing all the I/O messages through the operation switch. This implementation is different from primal files, for example, because of the fundamentally different ways in which we expect the object managers to be implemented. For devices such as line-printers, terminals and tape-drives, it seems realistic to expect that there will be one manager process per physical device. Unlike

[14]The description of the special operations on terminal devices is discussed in section 11.

[15]We might at some later date explore making some device managers clever enough to provide their own spooling, in which case one would be able to do frozen writes with the ability to cancel the writes. Such cleverness would likely lead to a number of special-purpose (spooling-oriented) operations, such as "perform output after a specific time", etc. While it might seem that such cleverness is more appropriately placed in a program and not in a device manager, for efficiency reasons one might desire to eliminate the middle-man. For example, a file to be spooled for printing, the requesting process, and the printer manager may all reside on different machines. There is little point in the data from the file to be passed through the network to the requesting program, then passed back through the network to the printer manager when the data could go straight from the file to the printer manager in the first place. Thus, a printer-object-manager may implement a "spool for printing" operation which takes the UID of the file to be printed as a parameter. Probably the act of spooling itself should be treated as an object and given it's own UID. Suggested operations on spool-objects: Create (to get a UID for subsequent transactions); Remove (to cancel a spooled action); TimeToBegin (to set the time for the spooled action to take place); as well as the usual printer-oriented operations (header format, font, etc.).

[16]Exotic as this may sound, it is easy to imagine a single manager for DEC-Tape drives and disk drives, for example.

the primal file system, which is accessed by many processes at one time, an individual device is typically a limited-access entity. Users typically require exclusive access to a device while they are using it. Thus we expect a device manager to be able to maintain a stream connection to everyone who wants to talk to its object. Very few constituent operating systems would permit a process to have so many open network connections supporting the message stream at one time, so we expect I/O from primal files to be datagram-based, rather than connection based. In contrast, I/O from devices may be connection-oriented, bypassing the operation switch for reasons of efficiency.

10.3.2. Reasonable defaults for unspecified options

In order to provide uniformity of access, the device managers assign reasonable defaults for their device-specific parameters (e.g., tape density) if the application program does not issue operations specifically setting them. The goal here is to provide an access mode in which the application program can remain largely unaware of the nature of the object receiving its output or providing its input.

10.3.3. Naming

Devices like any other Cronus objects have names in the globe Cronus symbolic namespace. These names may appear anywhere in the name hierarchy. For example, line printer devices are cataloged in the directory *:cronus:printers*, under names such as *imagen* for an imagen laser printer and *fifth_floor* for a standard impact printer located on the fifth floor. The symbolic catalog name is used only as a convenient means for accessing the device UID and plays no role in the way the Cronus system treats the device.

11. User Interface

11.1. Introduction

The Cronus user interface provides uniform, convenient access to the functions and services of the Cronus distributed operating system and the subsystems which run under Cronus. User requests for access to the functions and resources of the system are similar for all DOS components; that is, a request to run a program is the same no matter where the user access point is in the cluster, and no matter where the process that satisfies the request is run.

To date, we have supported Cronus access to users through a collection of commands implemented under the constituent operating systems of workstations and service hosts. This section describes a user interface which would be integral to the Cronus system, isolating the user from particular conventions of the individual constituent hosts, and allowing users to better exploit the distributed nature of the underlying DOS.

The user interface includes four major elements by which human users gain access and interact with Cronus to perform tasks:

1. The *terminal manager* is responsible for the behavior of the terminal or other device by which the user gains access to the system. Cronus supports a number of different terminal managers for users who have a direct connection to the cluster or who access Cronus through the Internet.

2. The *session manager* controls the user session from login to termination. It operates on the authentication data base (through the Authentication Manager) to verify the user's principal identity, and on the session record data base (through the Session Record Manager) to record information about the session. It also creates parallel execution threads and allocates portions of the terminal, under user control, to each thread.

3. The *command language interpreter* (CLI) receives requests from the user to create processes and execute programs to perform the tasks.

4. The *user programs* or *applications* that actually perform the tasks run in program carriers (see Section 5). The terminal manager, session manager, and the CLI cooperate in creating these processes, loading them, passing parameters to them, and directing the input and output to the places that the user has requested.

The design of the Cronus user interface has been influenced by the following considerations:

- The user interface should deal effectively with the *distributed* character of the operating system.

- Variations in cluster configurations and in user requirements will likely lead to a number of different user interfaces, and these interfaces will evolve. Therefore, the current implementation should focus on the underlying structural concepts needed to support a variety of presentation methods.

- The utility of Cronus depends on widespread accessibility. Therefore, the initial implementation should support commonly available terminals instead of more powerful devices which are now just becoming available.

- The user interface should support system reliability and error recovery from malfunctions during a user session.

The consequences of these observations for the design of the user interface in a distributed system are explored in the next section. The terminal manager, session manager, command language interpreter, and the pattern of the cooperation among them and their use of other system objects are discussed in the following sections.

11.2. Existing Interface Through COS

Access to Cronus is currently provided through commands implemented on each of the workstations and service hosts serving as access points. Terminal access is provided directly to these hosts and also through both the DARPA Internet and access points implemented on GCE processors. These components form the terminal manager component described in the introduction.

After establishing a connection to a host, the user will login to the system and to Cronus to establish a session. Under Unix, both registrations are performed by the same command; under other systems, where the system connot be easily modified, the user must execute an additional command to gain Cronus access control rights.

Thereafter, the command interpreter of the constituent host may be used to execute Cronus commands. The processes which perform these commands operate with the same Cronus access rights as the session. These access rights can also be changed, as needed, by executing appropriate commands.

Use is also made of window systems, available on the workstations, for presenting graphical interfaces in the case of the monitoring and control system, and for presenting "forms" based interfaces for general purpose command invocation tools.

11.3. User Interface Design for a Distributed System

The Cronus user interface is a generalization and extension of user interfaces provided by other computer systems. Since Cronus is a *distributed operating system* that integrates a collection of otherwise independent computer systems, the implementation of a function may be dispersed across the cluster. The Cronus user interface is independent of the user interfaces for the COSs.

The following are some of the design objectives for the user interface that have been influenced by the distributed nature of Cronus:

1. Command invocation and program control should be *uniform* across the cluster.
2. Multiple parallel activities should be supported directly by the user interface.
3. The user should be able to start and control distributed activities.
4. System operation should be independent of the location of the terminal manager, session manager, CLI, and user processes.
5. The user interface should support detection and recovery from malfunctions affecting only parts of a user's session.
6. The user should be able to issue commands directly to the COS.

First and foremost, Cronus itself provides for the uniform invocation of any command. The command interpreter finds the command in the Cronus symbolic catalog and creates a process for it. Because the symbolic name space is host independent, commands can be organized in any manner convenient to the user; for example, all the programs used to carry out a particular task can be cataloged in a private directory, even if some of them can only be executed on specific host types. The host is normally selected by examining the type of the executable file for the command.

A Cronus cluster may have more than one host of a particular type, and different copies of reliable files are stored on different hosts. The interface allows (but does not require) the user to communicate an intention to use a specific instance of any replicated resource.

A single user session may contain a number of independent tasks executing in parallel on different hosts. In such a session, the user can exploit the true parallelism which separate processing elements provides and reduce the effects of communications delays by selecting the host on which a task executes. Cronus provides device-independent mechanisms that support the use of a single terminal for controlling parallel activities. The effectiveness of a particular terminal for this purpose is, of course, dependent on the capabilities of that device.

A programmer can develop multi-part applications in which the individual parts can execute on different hosts. To the end user, the distribution of components can remain largely invisible, since the programmer and Cronus can take care of the details of the distribution. In particular, a task may consist of a multi-host pipeline of processes, in which a process running on one host can pass its output directly to the input to a process running on another host.

The Cronus architecture provides several kinds of access point. Although the user interface has comparable components for each of these access points, the location and mode of interconnection among the components will differ. The decomposition of function in the user interface permits flexible distribution of these components.

On the other hand, the distribution of the components increases the cost of synchronization and probability that a single host failure will affect the user session. To reduce synchronization traffic, Cronus does not maintain a centralized record of all elements in a user session. Rather, this data is distributed among the managers responsible for the individual parts. This makes the interface somewhat tolerant of failures and provides a basis for the design of a reliable user session.

The user interface facilitates direct access to COS functions through a user Telnet function, which can access the COS command interpreter for the hosts of the cluster. Telnet is treated as a parallel activity with other user activities; that is, it is a separate thread in the user session.

11.4. Overview of a User Session

A session begins when a user activates a terminal that is connected to Cronus and proceeds with a system login. The session normally ends when the user logs out. During the session, the user interacts with the system to run programs which interrogate and manipulate Cronus resources and to perform such job specific functions as word processing or data base inquiry.

Users gain access to Cronus in one of following ways:

1. Terminal access controllers (TACs). A Cronus TAC is a terminal multiplexer connected directly to the local area network. Cronus TACs are implemented in dedicated GCEs.
2. The Internet. The Cronus local network is connected to the Internet by means of an Internet gateway. Users outside the cluster may access Cronus through the standard terminal handling protocol (Telnet) which operates upon a lower level, reliable transport protocol (TCP).
3. Mainframe hosts. Cronus mainframe computers can have terminal ports that enable access to Cronus.
4. Dedicated workstation computers. A workstation is a computer that is, at any given time, dedicated to a single user. Workstation hosts have sufficient processing and storage resources to support non-trivial application programs, such as editors and compilers, and to operate autonomously for long periods of time[17].

[17]The Primal system will not support workstations.

The user interface has four principal modules: a terminal manager, a session manager, the session record manager, and the command language interpreter.

When the user activates a terminal, the terminal manager connects the user to a *session manager*. There is a session manager for each active user. It has a limited set of commands for initiating and manipulating sessions and session data. The login command, which initiates a new session, performs two basic functions. First, it identifies the user, establishes the access rights for the session, and gets the user data needed for session initialization. Second, it creates a session and records it in a *session record*. A complete description of the session is distributed among a number of system components, but the session record object records the existence of the session and certain other key items.

After the session manager has identified the user, it starts the initial subsystem specified in the user's principal object. This can be either a general purpose command interpreter or a special purpose application. The principal object may also request that the initial subsystem be run on a specific host.

The session manager maintains session data as part of its temporary state; that is, this information does not survive if the session manager crashes. The session record manager, on the other hand, maintains the basic information needed for session recovery in non-volatile storage.

The initial subsystem runs in the first processing *thread* in the session. The user may create more threads, each of which consists of a varying number of processes organized into a hierarchy rooted at the process created by the session agent. This program carrier is called the *head process* of the thread.

Often the head process is a *command language interpreter* (CLI). This is a program that interacts with the user to receive commands, which it performs by creating and controlling processes. In the following discussion, we assume that the head process of the current thread is the Cronus standard command language interpreter, which is called *cli*.

The head process can execute a command that terminates the thread. The session agent may also force the termination of a thread. The logout command terminates a user session. At the end of the session, the session record object is removed, and the terminal is free to support a new session.

Instead of executing logout, the user may *detach* from the session and re-attach to it later. Processes in a detached session are no longer controlled by the session manager and from the terminal. These processes will continue execution until they require terminal input or output, at which point they will block, and wait until they are re-attached. When the user re-attaches to this session, the new session manager and terminal takes over as the source of control and terminal input/output. The session manager command *resume* causes the processes to continue. This procedure is also used in recovering a session which has been detached by a host crash.

The user interface assigns the responsibilities for user session activities as follows:

- The *terminal manager* encapsulates the physical terminal device. It handles the terminal device, directs the keyboard input to the active process, receives the output from one or more active processes, and manages the display (for video display units).

- The *session manager* initiates user authentication, creates a thread, starts the initial subsystem, creates and manages additional threads, attaches and detaches sessions, and assigns terminals to processes.
- The *command language interpreter* reads and parses command line input, starts and controls processes that run the commands, and controls assignment of standard input and output.
- The *session record manager* creates and maintains records for active and detached user sessions.

In addition, other components of Cronus support the user session; of particular importance are the process manager, the catalog manager, and the authentication manager.

11.5. Terminal Manager

The terminal manager is the process which is closest to the user. It provides the Cronus interface to the physical device, through cooperation with the COS of the host to which the terminal is connected. The terminal manager supports three broad classes of device:

- *hardcopy* terminals that are strictly line-at-time devices capable of producing upper and lower case alphanumeric characters and the standard ASCII control character set;
- ASCII *video terminals* (often called CRT terminals or video display units) that support cursor addressing on a display screen that is large enough to support, for example, a full-screen editor; and
- *advanced* terminals (often called bit-mapped terminals) that contain a processing element and enough memory to support multiple display areas and graphical output.

The primary focus of the primal system is on the ASCII video terminal because there are many of them available today. Cronus supports the sharing of a single, physical terminal device among the parallel activities in a session. This terminal multiplexing will be most effective when an advanced terminal is available, but will be possible in a limited fashion with the other terminal types.

The terminal manager encapsulates the physical terminal; the corresponding Cronus object is of type CT_Physical_Terminal, which has a number of subtypes corresponding to the different kinds of terminals. One or more objects (called Cronus terminals or simply terminals in the discussion below) of type CT_Terminal is associated with each physical terminal. This provides a mechanism for multiplexing or sharing the physical terminal among a number of Cronus terminals. The Cronus terminal is the input/output device for a process. Since terminals are Cronus objects, they have all of the usual properties of objects, including host-independent access. In addition to the generic operations defined on CT_Object, the following operations are defined on objects of type CT_Terminal:

Open
Close
Read

Write
Activate
Deactivate

Programs may treat a Cronus object of type CT Terminal like an ordinary terminal, since it has a keyboard and a screen, although either or both of these may be inactive at any time. Each thread in a user session, and the session manager itself, has its own object of type CT Terminal, which will simply be called the terminal in the discussion that follows. Within a thread, processes coordinate their access to the terminal through the terminal manager.

If the physical terminal supports independent display areas (windows), the session agent maintains a window for status displays. The rest of the physical display contains one or more regions, each of which is used for the output of a single terminal. The physical keyboard can be associated with only one of the terminals at any time; the thread that owns this terminal is the current interactive activity in the session. The keyboard can be transferred to the session manager's terminal by a control character sequence. Once the session manager is in control, the user can execute commands to create new terminals, remove old terminals, and change the current interactive terminal.

Output to any of the regions currently displayed is immediately visible. Input is directed only to the current thread. Normally all input characters go to a single process. However, when one process creates another process, it may request certain (control) characters to be intercepted and sent to it; the interrupt facility discussed in Section 11.8 is implemented using this facility.

Processes invoke Read and Write operations on the terminal to get input from the keyboard and write to the display. These use large messages of indefinite length to provide a stream between the terminal manager and the process. A process will have two messages associated with the keyboard; it sends read requests on one of them, and receives the input on the other one. As keyboard input is collected, it is used to fulfill any outstanding read operation. Since the terminal is shared among the processes of the thread, characters are sent only in response to a read request. If there is no outstanding request, the terminal buffers characters until it exhausts the space allocated for them.

When control of the keyboard is transferred from one process to another, the old process stops issuing read requests. If the new process needs keyboard input, it establishes the two messages used for the stream and begins issuing read requests of its own. The PSL routines for reading and writing take care of the details of establishing the messages, so ordinary applications need not be concerned with them. The Write streams are not controlled; simultaneous output from two processes in a thread may become interleaved unless they are coordinated by the application program logic.

Each terminal has mode settings which control its behavior. Among the most important are the following:

1. Read activation termination character set: An input character from this set terminates the current read request. The terminal manager stops sending characters after it transmits the ones it has, including the termination character, until it receives another request.

2. Echo control: Input echoing at the terminal manager may be either on or off. If it is on, it may be performed immediately or deferred until the characters are used to satisfy a read request.

3. Buffering and local editing: Terminal input may be buffered until an activation request termination character is typed. If the input is buffered, local editing functions are also available. If the input is unbuffered, it is sent as soon as it is accepted when a read request is active; the application process then assumes the responsibility for editing functions.

4. Interrupt character handling: The user may set certain characters as interrupt characters; see the discussion in Section 11.8.

11.6. Session Manager

The session manager creates and removes user session records, controls the allocation of the physical terminal display, and creates and controls threads.

During a simple session, in which a user executes a series of commands sequentially, the session agent is largely invisible to the user. The user may, however, wish to initiate and control parallel activities. Each collection of parallel activities is a *thread*. Session threads are objects of type CT_Thread. At any time during the session, the user can instruct the session agent to create additional threads which operate in parallel with other existing threads[18]. A thread can be used to support parallel processing or to maintain the state of some activity while the user shifts attention to another activity.

The first process created in a thread is called the *head process*, and is usually a command language interpreter. The default head process is an instance of the principal's initial subsystem, but the user may select any program in the Cronus symbolic namespace.

A new thread is created whenever a Telnet connection is opened, with the Telnet process at its head. The connection may be to any Internet host, either within or outside the cluster. For the foreseeable future, Telnet paths to cluster hosts will be needed to support activities not yet incorporated into Cronus, such as maintenance of the COS.

The following commands are supported directly by the session manager:

- Start a new session (login)
- Terminate a session (logout)
- Attach session agent to an existing session (attach)
- Detach session agent from an existing session (detach)
- Initiate a parallel activity (create_thread)
- Terminate a thread (killthread)

[18]There is user-settable control key that activates the session manager so the user may invoke session manager commands.

- Create a Cronus terminal (make_terminal)
- Remove a Cronus terminal (remove_terminal)
- Map thread to region (map_thread)
- Display threads (showthreads)
- Activate named thread (thread)
- Telnet to host (telnet)

11.7. Session Record Manager

The session record manager maintains the centrally accessible, non-volatile record of active Cronus sessions in objects of type CT_Session_Record. A session record object contains the following data:

- Session UID
- Creating principal
- Time of Creation (for identification purposes)
- Lists of thread UIDs
- ACL
- Session agent ProcessUID

A session record is created at the beginning of each Cronus session. During the session's lifetime, data is added and removed by the session agent. The session record is used in recovery after a host or system crash.

The session record can be accessed by other programs to report about an individual session or all current sessions. In addition to the generic operations, the following operations are defined on objects of type CT_Session_Record.

- Read_Public
- Read_Private
- Write_Session_Record
- Lookup_Principal

11.8. Command Language Interpreter

A user request usually consists of a command name plus one or more *parameters* or *arguments*. There are three basic kinds of arguments for *cli*: names of objects from the Cronus catalog; control parameters, called *switches*; and application-specific parameters. Switches may be associated with either the command as a whole, modifying its behavior, or with one or more of the object names that appear on the command line.

If one thinks of the command as a series of words typed on a line, the command name is the first word. The command name specifies the action to be performed; the actual name is often a simple English word suggesting that action, for example, *print*. *Cli* interprets the command name as an entry in the Cronus symbolic catalog: it expects the command name to be the symbolic name of an object of type CT Executable_File. Either a complete or partial pathname may be entered on the command line. A designated set of Cronus directories (called the *search path*) are used to resolve partial pathnames; the first match encountered causes the search to stop.

There is a small set of commands built into *cli*. These are used to control the command interpreter's environment (such as the current working directory) and the execution sequence of commands. Executable objects may be either process *images* or files containing commands. The built-in commands that control execution sequence are most often used in command files.

The executable object may be augmented by a syntax definition, so the command interpreter can know the number and type of the arguments, default and legal values for the switches, and help information for the command. Users may associate private syntax definitions with public commands. Commands which have syntax definitions, either private or public, are called *defined commands.*

Command arguments are passed as part of the process descriptor of the new process. When the command syntax definition is available, *cli* performs type and range checking on parameters, and conversion from alphanumeric to internal representations for certain of types, including Cronus object name and integer. Both forms are passed to the application process, since the character string form is of use in some cases, for example in generating error messages.

The syntax definition facility is particularly valuable in a distributed environment for the following reasons:

- The cost of remote command invocation is generally higher than it is in monoprocessor cases. Parameter error checking reduces the frequency of execution failures caused by usage errors.
- If the command interpreter knows the names of some of the objects that the command is operating on, it may be able to use object location as one criterion in its selection of a site for command execution.

Many command arguments are cataloged objects. Cronus supports a *working directory list*, which is an ordered collection of directories that are used in relative pathname searches for named objects. The user may change this list at any time. The *cli* also supports partial name recognition. The user may press a key to get a list of all matches for the partial name, using both the working directory list and the standard wild-card facilities of the Cronus catalog, from which the actual name may be chosen. There is also a deferred recognition key which allows the user to ask for the matching to be done, but not reported interactively.

The help key can be used to display help information which is found in the syntax description of a defined command.

The command interpreter allows a user to provide a host designator when specifying an object name, including the name of the command itself. For example,

edit textfile@CVAX

would invoke the editor on the copy of *textfile* stored on the Cronus VAX,

copy file1 file2@GCE3

would make a copy of *file1* under the name *file2* and store the new file on host GCE3, and

Radar@CLXX other parameters

would select host CLXX to run the subsystem Radar.

Objects of various types may be cataloged in the Cronus symbolic name hierarchy without restriction. Often, a user will wish to select objects of a specific type, so a standard switch is defined for type designation. As an example, a user would type

dir_display file_name.*/type=reliable_file

to display the names of those objects in the current working directory list that match the wildcard pattern file_name.* and are of type CT_Reliable_File.

Cli performs two kinds of initialization. First, internal variables are set from a profile data file, which consists of lists of (name, value) pairs. This file can be maintained using *edit_key_value*. Second, *cli* executes a profile command file.

After *cli* has collected and parsed a command, it creates a process, loads it with the executable image and starts it. Normally the process uses the same terminal as the command interpreter does. Therefore, *cli* releases control of the terminal to the user process, and waits for it to terminate before collecting another command.

Cli uses the process support for input and output redirection. The redirection is indicated by the punctuation character >, thus the command

dir_display file_name.* >newfile.lst

would place the result of the catalog lookup of file_name.* in the file newfile.lst. When *cli* redirects output into a file whose name did not previously appear in the Cronus catalog, it creates a new primal file. The user may use the standard switch (/type) to designate another type, for example,

dir_display file_name.* >newfile.lst/type=reliable_file

will create a reliable file to receive the output.

The user can specify that two or more commands should be executed simultaneously and linked together linearly, in such a way that the output of the each command becomes the input to the next one. We refer to the collection as a pipeline. Since the components of a pipeline may be on different hosts, the user can dynamically construct multi-machine distributed commands.

11.9. User Processes

In most cases, actual work of an application is carried out by a user process that is created in response to a command issued to *cli*. Application programs typically make extensive use of the PSL. In this section, we discuss interrupts and user error reporting, both of which are supported by the PSL.

Sometimes a process needs to be terminated by an *interrupt* or *signal*. Cronus supports two forms of interrupt: a *hardkill*, which terminates the process immediately without giving it the opportunity for application-specific termination processing, and a *softkill* that gives the application process the opportunity to terminate cleanly. In the event that programs do not respond to softkill requests, hardkill can be imposed. Interrupts are usually invoked by typing a control sequence during a user session, but they are also generated by a command.

Programs may choose to receive softkill signals, and use them for application-specific purposes unrelated to process termination. *Cli* will always receive the hardkill signal and remove the application process.

Interrupts invoke the Stop operation on process objects. The exact implementation on a particular host depends on the facilities of the COS that are available to the process manager.

The processes created by *cli* form a hierarchy of process objects, which may be decomposed into sub-hierarchies of the thread object. Any subtree of the thread hierarchy is called a *process group*. An entire thread is the largest process group. Process groups are managed by the program carrier manager in the current implementation. Operations on process groups support convenient control and cleanup of process subtrees.

Methods for reporting errors in Cronus are designed to support a variety of program structures and execution environments. There are two basic program structures:

- Asychronous processes, often called manager processes because object managers are of this class; these processes receive messages from a number of sources and may not wait if they issue requests to other managers to satisfy incoming requests. Error handling in manager processes is discussed in Section 4.6.
- Synchronous processes, which process data that arrives in a more or less predictable fashion, often from a terminal or a file. When these processes send messages, they usually wait for a reply.

We have identified the following execution environments:

- Independent processes are asynchronous processes, particularly object mangers that are daemon processes started by the Monitoring and System or by another daemon process.
- Interactive processes may be either synchronous or asychronous. In this environment, a human user carries on a conversation with the process. Examples of processes in interactive environments include the traditional applications of distributed systems: multi-host database systems, office automation, and program development systems.
- Pipelined processes consist of two or more programs which might normally be run in an interactive environment that are connected in such a way that the upstream process writes its output on the input of the downstream process. A pipeline can span host boundaries.
- Background processes are generally interactive programs which are set into execution in such a way that the data which normally comes from the user is found somewhere else (usually in a file).

In the interactive case, where the error is reported directly to the user, we have a situation that is similar to the one in an ordinary, centralized operating system. It can be seen that error handling is similar in pipeline and background cases.

A program in an interactive environment will also report certain errors to the Monitoring and Control System (MCS). These include errors caused by system resource limitations and some kinds of access control violations.

Independent processes, including Cronus managers, report errors to the client which issued the original request, and may also send a message to the MCS. In addition, Cronus managers keep statistics on the kinds of errors which have been detected, and report them to the MCS periodically.

The responsibility to terminate or continue processing belongs with the application or manager, so PSL routines never take preemptive action, and never terminate the process. The PSL routine cannot understand the situation well enough to exit properly, since the routine may be executed within an atomic transaction, or within a composite action which has other work-in-progress entries (see Section 4.6). Instead, it sets parameters describing the condition in an error block, and the application error handler fields the error and processes it.

The standard error list may be found in the general Introduction to the Cronus User's Manual. Each PSL routine page in Section 2 of the Cronus User's Manual lists the errors which may occur during the execution of the the function. In most cases, an interactive process would perform any necessary cleanup, and then use the standard error reporting routines.

Whenever an error is detected in processing a request from a client process, the error condition is reported through the reply message. The error procedure uses the standard message structure, and certain assigned keys. When it is necessary to report an error to the MCS, the process uses a standard routine to generate the message to the MCS.

12. Monitoring and Control

12.1. System Capabilities

The Cronus Monitoring and Control System (MCS) provides the functionality of an operator's console. From any suitably controlled access point, the operator can examine the status of the cluster's resources, invoke operations changing the state of the resources and resource management policy, and view the effects of those operations. The operator can evaluate long term system reliability and compliance with resource management policy by reviewing logs of status data kept by the MCS. Resource managers may submit event messages to alert the MCS of irregular events. If an event requires operator attention, the message will be displayed on the operator's console. Otherwise, the message will be recorded and available for later review.

The Distributed Operating System (DOS), as viewed by the MCS, can be divided into three layers. At the bottom is the constituent resource layer consisting of processors, peripheral devices, network substrate, gateways, Constituent Operating Systems (COS) and network protocol support. Above that is the Cronus support layer, consisting of the Cronus kernel, Cronus Interprocess Communication mechanism (ICP) and the Cronus services managing constituent resources. Finally, at the top, distributed application programs are built from collections of processes and managers.

The MCS focuses on the needs of problem diagnosis and resource management. The implementation emphasizes support of the Cronus layer, the managers, and the resources they provide. Since the set of services is extensible, the MCS is designed to accommodate new services. The MCS forms the basis for monitoring the application layer. The MCS also provides operator interface, configuration management, data collection and process coordinations facilities that can be employed by other services.

The MCS provides some direct access to COS facilities, but such support is limited by our desire to modify the constituent host software as little as possible. The operator can discover which hosts are up and can cold start or halt the Cronus kernels. This requires support by the hosts of a non-Cronus protocol for starting the Cronus kernel, possibly downloading the kernel image for diskless nodes. Once Cronus is operating, the MCS communicates with managers that provide the interface to the constituent resources. This hides the differences between the constituent resources and the implementation details of the interface software from the MCS.

Failure of the MCS or its operator must not endanger essential DOS services, although the performance of some Cronus services may degrade. Essential functions, such as manager restart and resource management, are performed by cooperating managers. The MCS role is limited to adjusting resource management policies, to improving the reliability of the Cronus services, and to providing a diagnostic access point for the operator. The MCS itself is a distributed application program split into separate managers. The components may be reliable and use replicated data when appropriate. The operator station is not bound to any particular site, although certain information gathering functions are most conveniently performed at one location and certain control functions are subject to access control.

The MCS supports automatic processing to enhance system reliability and regulation. It can monitor a collection of values, detect particular conditions, and then perform a prescribed action, such as restarting hosts and managers when they crash. Or, the MCS might alert the operator when 90% of the disk space managed by a particular manager had been allocated; the MCS can then automatically arrange

for file creation requests to be routed to other managers. In this way, the MCS is used to evaluate experimental algorithms which will then be moved to the managers if they are effective or discarded if they are not.

We are not initially concerned with issues of multiple clusters or very large clusters, although we are sensitive to scalability. As the monitoring domain grows we expect to divide resources into overlapping regions, where resources whose behavior strongly interact are in the same region and resources whose behavior is typically independent can be placed in different regions. A regional monitoring center will them monitor each region and will exchange summary information with other monitoring centers when more global information is needed. As we said, this is beyond the scope of the initial version.

12.2. Sample Scenarios

12.2.1. Problem Diagnosis

Most problems are reported by users when a command fails or behaves irregularly. The operator must determine whether the command is in fact misbehaving and if so, what is causing the problem. This is done by comparing the expected outcome of an operation with the actual outcome and trying to discover the cause of any deviation.

For example, a user may report that he can't access a file that he normally uses. This problem can occur if the user's privileges have been changed, if the file has been deleted, if the access control list to some part of the file's pathname has changed, if the file manager or host where the file resides has crashed, or if one of the directory managers that catalogs the file and its pathname has crashed. Intermittent failures can occur when a manager, a host or the network is saturated. During development, bugs can cause managers, hosts and the network software to enter states where they appear to be available but do not respond to all requests.

The MCS must allow the operator to examine all these symptoms and possible effects from a single console. The operator first tries to repeat the user's operation with the user's access rights. If that fails, using special privileges, the operator checks to see if the file exists. If the file does exist, the operator must repeat the user's command and trace its execution through the system; this requires a little understanding of how the system works. To lookup a file we first locate a catalog manager, then find the UID of the file represented by the given name, the locate the file and finally open the file for reading or writing.

Each of the managers involved keeps a log of the operations it has performed. The amount of detail kept in the logs can be varied by the operator. The MCS allows the operator to examine these logs in order to trace the progress of the request. Using the logs the operator can determine which managers processed the request and where the request either got lost or was rejected. The operator can then invoke commands targeted to specific managers to further localize the problem.

12.2.2. Resource Management

The operator uses the MCS to review the system's behavior and to evaluated how well the system complies with chosen resource management policies. Most of these policies are vaguely described, different applications require different policies, and different policies conflict. Examples include balancing resource consumption, minimizing average response time or ensuring priority access to resources by privileged users.

The operator adjusts policy parameters, such as resource quotas, cache sizes and routing priorities to affect the resource management decisions made by the system. The services combine these policy parameters with measures of actual resource usage to decide as where to place new object instances and to route requests for processing.

Polling intervals are automatically adjusted to ensure that the effects of the change will be properly sampled. The operator then reviews the historical data to evaluate the effects of the change. Graphical presentation is especially important for quickly identifying trends and resource distribution.

The resource management decision making process is not well understood. Our goal is to provide the mechanisms and tools to handle experimentation, to prevent chronic saturation of parts of the system, and to discover causes of chronic saturation when it does occur.

We identify three degrees of resource contention: none, moderate, and saturation. Each of these situations require different handling. When there is an adequate supply of a resource and the resource is fairly homogeneous in all its instances, we don't need to worry about resource management. We may allocate any available instance to satisfy a request. When contention begins to occur, we have to consider where to allocate the initial instance of the resource. This decision involves considering the cost of the resource, the cost of accessing the resource, and the cost of moving the resource later if a bad choice is made. We expect that this can be done by the system, with the operator periodically adjusting parameters that regulate the decisions. When the supply of a resource is nearly exhausted we need operator intervention to correct the situation. Generally, eliminating the saturation will require either increasing the supply of the resource by activating new processors or disks, eliminating some users of the resource by stopping application programs, or rearranging the placement selected instances of the resource. These decisions require an understanding of the intended use of the system and priorities among the uses that the system cannot handle by itself.

12.2.3. Performance Evaluation

How much does it cost to create a file, measured as some combination of application waiting time, of processor and operating system time for the managers that are run to service the request and of network use to request and coordinate the file creating? This is an important issue we need to improve system performance and need to discover where the time or resources are being consumed. This information can also be used when we have to charge system users in order to recover the cost of constituent resources, but this is not a goal of the current system.

The monitoring itself does not greatly increase the cost the normal operations. Also, in performance evaluation, we are often combining heterogeneous measures of cost, such as time and space usage, to produce a measurement of user satisfaction. This requires assigning relative values to each which may or may not reflect the actual user preferences. Also, in performance evaluation, it is not always clear what low level events and constituents are the major sources of the cost at higher levels.

This information can also be used to guide resource management decisions. Using a model of the cost of performing an operation, the system can make resource management decisions that it expects will have acceptable costs in future decisions.

12.2.4. Experimentation

The MCS may also be used to monitor DOS experiments and the objects that may be introduced into the system to implement that experiment. The MCS will be integrated with the manager developments tools to simplify the cost of introducing monitoring to a new manager.

12.3. Structure of the MCS

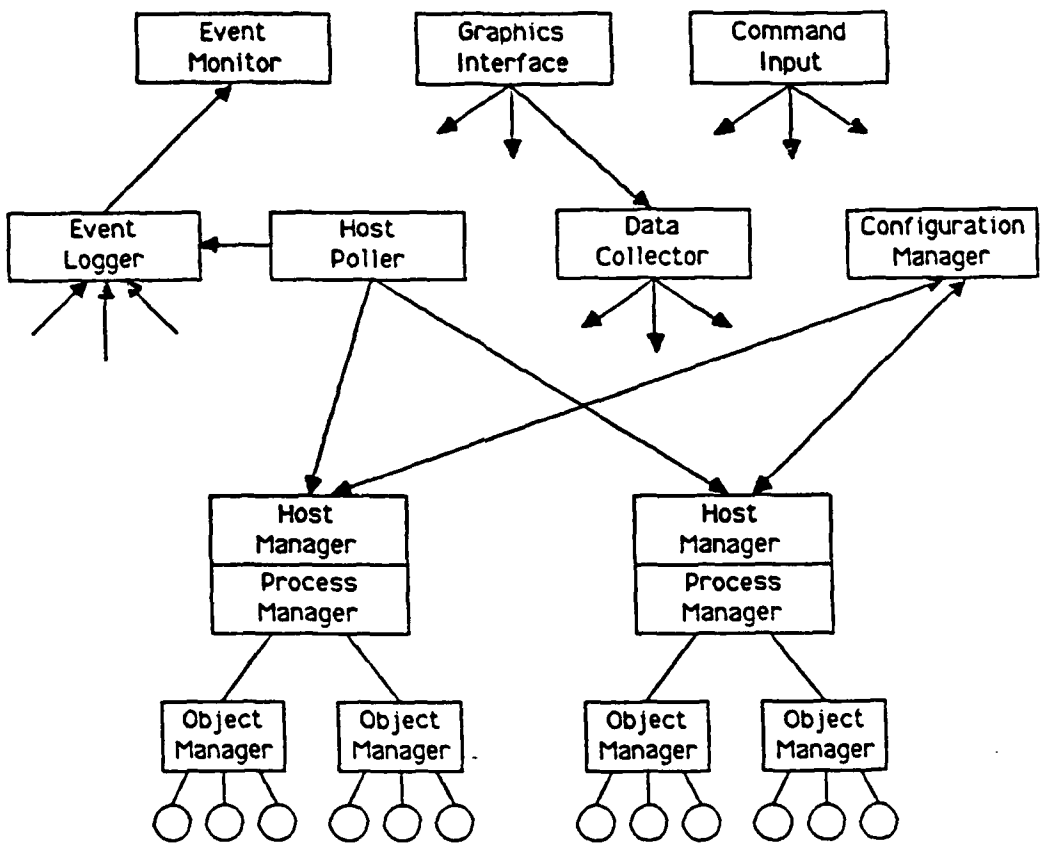
The MCS performs configuration management, event logging and reporting, host availability monitoring, and data collection, and provides an operator interface for data review and command input. These functions are implemented by a collection of cooperating Cronus processes and probes into the managers being monitored. The relationship among these components is displayed in Figure 12.1.

12.3.1. Configuration Management

The configuration manager provides a logically centralized service for controlling the placement of managers. When a developer creates a new service, he also creates an associated service data object. The service data object lists the object types supported by the service and identifies the person or group responsible for maintaining software associated with the service.

Placement of managers that support the service is done by manipulating host data objects. For each known host in the cluster, a host data object is created. Each host data object lists the services running on the host it denotes. A manager may be assigned to run on a particular host by adding a reference to the appropriate service data object to this service list.

The configuration objects are managed by a configuration manager. Access to the objects is regulated by the standard Cronus mechanisms independently for the service and host data objects. Customerily, developers will maintain the service data objects, and system operators will maintain the host data objects.



MCS Architecture
Figure 12.1

The Cronus kernels acquire the appropriate information by requesting it from the configuration manager, either when the system is rebooted or when a client submits an update command to a kernel. The request submitted by the kernel to the configuration manager identifies the kernel's host address; the configuration manager will then search for the appropriate host data object and construct the service list. The service list will then be sent to the kernel is pieces, each small enough to fit into a small message to minimize the amount of underlying supported needed by the kernels at cold start time. The kernel's record the information locally, on their host, in stable storage, and will use the locally stored information if the configuration manager is not available at a later time. Since the configuration manager may be replicated for reliable operation, we do not expect this information to be needed very often, except when restarting large portions of a cluster.

12.3.2. Event Logging and Reporting

Event reports are submitted to the MCS to describe irregular events. For example, Cronus kernels report manager crashes and restarts and the host poller reports host crashes and automatic restarts. This mechanism can also be used to report when a file manager runs out of space or when someone is trying to log in but has repeatedly entered the wrong password.

Event reports are handled by the combination of an event manager and an event monitoring program. The manager maintains objects that are determine how events are collected and filtered for logging and display. The monitor program is used by the operator to review event reports as they arrive. Additional monitor programs can be written to automatically correct problems when they are reported.

Event reports include a written description of the problem, intended for operators. A severity code can be included to indicate whether the report is just for information or whether a problem arose, and if a problem arose, whether it has been automatically corrected. The reports optionally include a numeric code identifying the problem and the UID of the object that was affected by the event. These values can be used by automatic monitoring programs to determine what actions are needed to correct the problem. Event reports also identify who is reporting the event, so that further information can be requested.

The event manager maintains two kinds of objects: event collectors and event filters. Event reports are submitted to the collectors; the generic collector object is used for reporting system events, other collectors may be created for use in services or in applications. Event filters determine how event reports are handled. An operator attaches collectors to filters; events reported to any of the attached collectors will be forwarded to the filter. An operator may also describe a filter to select which messages will be accepted by the filter or which messages. Events which are accepted by the filter will be optionally recorded in a log file.

The event monitoring program connects to a set of filters to monitor the event reports they accept. Thereafter, whenever an event report is accepted by an event filter, a copy of the report will be forward to each monitor that is connected to the filter. When the monitor receives the operator is alerted and the message is displayed.

12.3.3. Host Availability Monitoring

The availability of hosts and of Cronus on those hosts is monitored by a host poller manager. This manager is responsible for determining which hosts are attached to Cronus, monitoring whether they are available, and reporting any changes in availability to the operator. This manager does not monitor the availability of managers for Cronus services--that is the responsibility of the Cronus kernel.

The host poller periodically updates its host list by broadcasting a request for all hosts to report their status. A *host poller* object will be created for each newly detected host. This collection of objects forms the host poller list. Each object records the status of the host it denotes and provides polling parameters, such as polling period, that the operator may adjust. Once a host is detected, it will be remembered indefinitely, regardless of availability; only an operator can remove the poller object.

Using the host list, the host poller periodically checks to see if each host is still available by individually asking the host's status. If a host fails to respond, the failure is reported to the system event collector. After several failures, the host is assumed to be down, the poller discontinues polling of the host, and reports the crash to the system event collector.

For host that support remote restart, the host poller can attempt to restart the host. This is optional, controlled by the operator. The operator selects whether restart should be performed and which of several procedures should be used to initial the restart. If restart has been selected, the poller will make one attempt to restart the host; if it fails, the operator must correct the problem and initiate the restart.

Monitoring of Cronus availability is performed using Cronus IPC. A special "are you there?" protocol is supported to allow the MCS to determine whether a host is available even when the associated Cronus kernel is not responding.

12.3.4. Status Data Collection

Status data dynamically describe system resources. These resources include active components such as processors and the network, resources such as file space and line printers, and Cronus software components, such as managers and application programs. The data monitored for resources describes availability, location, load and access time. Averages, standard deviation and rates should also be available. Policy and resource management data is reported. Cost information for performance evaluation is provided.

Managers report status data to the MCS in response to a poll request. This allows the MCS to control the data collection process, varying the set of data collected and collection intervals depending upon what the operator is examining and what the MCS is doing, and does not burden the managers with the need for additional mechanism to ensure that the data is periodically reported. The MCS temporarily increases the polling frequency for managers that are affected by a command invoked by the operator. The polling interval may also be reduced when the MCS notices activity on a particular manager. Also, the operator may specify a fixed polling interval, or request an immediate poll of a particular manager.

Much of the polling is performed by directly contacting the managers responsible for the object whose status is to be retrieved. Broadcasting, by itself, is not adequate for issuing the poll requests since delivery of broadcast messages, although likely, is cannot be guaranteed. This becomes a particular problem when a host is heavily loaded, since it is then that we are most interested in it but it is most likely to drop broadcast messages. Also, broadcasting does not allow us to regulate the sampling interval for particular managers. Broadcasting will be limited to locating newly restarted hosts.

12.3.4.1. Status Reporting

The most commonly used status report request is *report status*. The managers for most services support this request. The status data managers return varies from one service manager description, resource description, health and availability information, traffic statistics, constituent resource consumption and resource management parameters.

The report describes a manager by giving its type name and type code, process ID and host address. Host managers will include a host name. Processes will not include the type information. Access rights and other parameters of the process can be gotten with the "get process parameters" request.

Each manager lists the resources it manages. A process manager would list the processes and their names. A file manager would list file systems and for each give the capacity and amount currently being used.

The fact that a manager replies, indicates that it is available. However, it may be currently refusing a subset of the operations it customarily supports; this would be indicated in the report status reply. Also, some of its resources may be unavailable or partially allocated. For each resource, the total capacity and current consumption are listed. For example, the size of each file system and how much is allocated to files and index blocks would be listed by the file manager. For IPC, the last time a message was sent to and received from each host might be given.

Traffic statistics are given for the manager and for each resource. This includes the number of operations performed by the manager, such as I/O operations, file opens, and so on. For IPC, the number of messages and octets sent to each host would be given.

The constituent resource consumption is given for the manager, each resource provided, and for each class of request services. This gives processor usage, process size, disk usage, I/O activity, how long ago it was started, and any other relevant cost information needed for performance evaluation of the manager. This is itemized for each resource managed. For example, the process manager would list how much memory each process consumes, how much I/O and paging activity, how much CPU time it had consumed, and how long it has been since it was started. In giving constituent resource information we must remember to normalize figures to account for the heterogeneity of the hosts. Space on systems is managed in a variety of units of size: bytes, blocks of 512 bytes, 1K bytes, 4K bytes and others. We must be careful either to convert to known units or specify the units in all cases. Clocks are not necessarily synchronized so times must be relative to a particular host.

Finally, the parameters used to make resource management decisions are given. Some of these are constituent resource consumption values already mentioned. Others are policy parameters specified by the operator or MCS to regulate the resource management behavior. Any decisions made by the manager, such as deciding that all create requests should be refused, will also be indicated in the status message.

Following are specific examples of the kind of data reported. The actual information supplied in the packet will be driven by are needs as Cronus is developed. If the message size becomes be too large for a single packet we will divide the data into multiple requests based on the kind of data. Also, we may introduce commands to vary the amount of detail reported since complete detail is not always needed and since most data is never examined by anyone.

The operation switch reports the following information for communication with each host foreign to itself, and each manager local to its host:

- The foreign host name or local manager UID
- The number of bytes and messages sent and received
- The first and last time a message was sent or received.

The process manager reports:

- Process capacity
- Active manager count
- *Active non-manager process count.*

For each process the process manager reports:

- The Cronus process UID
- The local host process id
- The process name
- The object type if the process is a manager.
- The image used to load the process
- The time the process was created.

It will also report any additional statistics, such as processor usage or paging activity, that can be supplied by the COS running the process.

The primal file manager reports:

- The number of open files
- The number of disk accesses
- The time spent processing requests
- The total disk space managed
- The amount of disk space available.

The Fast files and COS files will supply the same fields, however some values may not be available in particular implementations.

The directory manager reports:

- The dispersal cut pathname
- The number of entries each above and below the dispersal cut
- The number of directory references each above and below the dispersal cut.

The authentication manager reports:

- The number of authentication requests processed
- The time spent processing requests
- The number of confirmed requests.

12.3.4.2. Data Archival

Archives can be stored either in the COS of the individual managers, or collected and stored by a group of archive managers. We will initially collect the status data and store it in one place. This will simplify data retrieval development when our major concerns are with host to specify which data items to retrieve rather than how to find all the stored data files. If the network traffic required to support the centralized log file is unacceptably high, we will store the logs with the individual managers and develop a distributed retrieval mechanism.

Since the amount of data can grow indefinitely, methods for discarding obsolete data or retaining only a periodic sampling of data are required. Data may be archived on tape before deletion. We will also require key oriented retrieval methods. This can be accomplished by periodically copying the recorded data and the associated keys into a data base management system.

12.3.4.3. Data Analysis

The analysis portion has two functions: combining the data from various sources to produce summaries and discover trends; and monitoring the data to alert the operator when particular events occur.

12.3.5. Operator Interface

12.3.5.1. Windows and Menus

Three types of windows are used to display information to the operator: MCS status; resource status; and event reports. These are distinguished because the operator handles each kind of information differently. MCS status is used only when changing views or invoking commands. Resource status is used to examine the status of the cluster and is used most frequently. Event reports should be displayed along with either a visual or audible alert to attract the operator's attention. Event reports should be recorded so the operator can view them in order or review previous reports.

Commands are typically invoked on an object of the status display of an object by selecting the object and then selecting the command from a menu that appears. This reduces the information the operator must remember about command protocols and formats. Other menus allow the operator to change MCS parameters.

12.3.5.2. Hierarchical Information Access

Data display is organized in a network of status views. The operator begins with views that summarize the status of a service. For example, a summary of the file managers would show how much space is being managed, how much of it is being used, how many requests have been serviced, how many file managers are active, what is the mean time to failure of an average file manager, etc. From there the operator can move to more detailed views. For example, a view giving the same information, but showing the values for each participating manager, or showing what percentage of the resource each manager handles or what percentage of the requests each manager services. Or the operator might choose between views designed for reviewing resource management and views designed for evaluating system reliability. Additional detail on any particular item can be displayed by selecting that item and invoking a display command.

12.3.5.3. Graphical Presentation

There are three uses of graphics: quick recognition, trend projection and comparison. Distinctive icons, distinguishing either the object or its function, are used to display objects or functions that the operator will need to locate quickly. Diagrams show the relationship between objects, such as traffic flow. Graphs allow the operator to evaluate average system behavior and project trends of future performance. Charts simplify comparing performance, load and resource consumption in different parts of the system. Values that have associated thresholds are displayed on gauges so the operator can quickly recognize when the thresholds are being violated.

In addition, cues such as size, color and image reversal will be used to guide the operator in locating important display objects. For example gauges whose thresholds are exceeded and switches for managers that have crashed can be colored red to attract the operator's attention. Hosts and managers that are rebooting and other situations where an important operation is in progress can be colored yellow.

12.3.6. Control

The coordination and control functions of the MCS consists of a very low level module and a higher level module. The majority of the MCS uses the high level module, a Cronus service that communicates with its probes using Cronus IPC. The low level module uses only the lowest level of network protocol, such as a user datagram protocol. This primitive low level can be relied upon when little of Cronus is functioning. It provides the functions required to bootstrap Cronus, to examine and alter memory on Cronus hosts and to do simple monitoring of the Cronus network.

Access control for the high level commands will be handled by the Cronus IPC. Access control for the low level commands will be limited, initially requiring no more that a password to be submitted with the request, or using an access control list of known physically secure hosts.

Control of the cluster is organized hierarchically. The MCS is directly responsible for the Cronus kernels running on the hosts. The kernels then share responsibility for their own reliability and the reliability of the managers running on their host with the MCS. The MCS communicates directly with the managers to get status data about the managers and the constituent resources they provide. The MCS has essentially no direct communication with the resources provided by the managers except during cold start, when the managers are unavailable.

12.3.6.1. Cold Start and Forced Shutdown

We assume that when a host on the Cronus cluster is booted, it will automatically load the Cronus kernel. The kernel will then notify the MCS through the system event collector. Hosts that do not store the kernel image locally notify the host poller when they are restarted and then wait for a kernel image to be downloaded. There may be a few hosts, due to physical limitations, which can neither start themselves nor notify the poller of their presence. The host poller will maintain a static list of such hosts and periodically poll for their presence, reinitializing them when appropriate. This allows the MCS to automatically build a host list. When the kernel receives the restart command, it starts the primal process manager, which, in turn, starts a selected set of managers. The operator can specify that a host is self restarting, in which case it does not await the restart command.

Restart of the MCS itself after a system crash should be automatic too. Manual restart requires starting the Cronus kernel and managers on the hosts and then starting the MCS component processes. The MCS then broadcasts requests to determine which hosts are available with Cronus kernels loaded. The operator then has the option of letting the MCS bring up the cluster or of manually bringing up the hosts one at a time.

The operator can also force a Cronus kernel to halt without using Cronus IPC. The routines performing this command should also ensure than all managers on the host have been halted too. This is needed to restart hung kernels and sometimes to clear network problems. When possible, using the command should produce a diagnostic dump of the kernel for use in debugging. Booting Cronus after a forced shutdown requires a cold start command from the MCS and possibly downloading a new kernel image.

12.3.6.2. Restart and Cronus Shutdown

The operator can invoke operations on the Cronus host manager to terminate the kernel. These commands can either terminate the kernel permanently or terminate just the managers and leave the kernel waiting for a restart command from the MCS. The permanent shutdown requires reloading the Cronus kernel before a restart command can be processed.

12.3.6.3. Creating and Removing Managers

Any manager can be started or stopped by sending a create or remove request to the process manager of the selected node. A manager that has been removed will not be automatically restarted. We assume that the action was deliberate, unlike crashes which are usually unintentional.

12.3.6.4. Resource Management Policy

The MCS can change policy parameters that influence resource management decisions. The major effect of resource management it to choose the placement of new object instances and where resources will be allocated to service particular requests. The values of these parameters will be reported in response to the MCS polling requests.

12.3.6.5. Set Logging Level

The operator can vary the amount of detail that is recorded by managers in local event logs. This command also varies the amount of detail send it event reports submitted to the MCS by the managers.

13. Application Development Facilities

13.1. Introduction

Object-oriented programming simplifies the design and implementation of Cronus system components by capturing the essential characteristics of a problem, and hiding complexity of its implementation behind the operation interface. The Cronus Object Model is equally useful for systems and applications programming in Cronus, and so it is anticipated that many Cronus application programs will be constructed using the techniques that have been used for systems programming in Cronus. To make programming easier for applications developers, software tools that aid and automate the development of distributed applications have been developed.

This section describes the implementation of the current set of programming tools towards simplifying the development of object managers by automating the implementation of their common parts. Many of the details of implementing a distributed application have been hidden by these tools, allowing the application developer to concentrate on the implementation details specific to his problem, and leave the difficult aspects of distribution to the tools.

The features of the Cronus application development facilities are:

1. **Asynchronous Request Processing**: Object managers developed using these tools are able to process multiple requests simultaneously. This capability is accomplished by using a non-preemptive, coroutine-style task facility to share the manager process' computation among concurrent request processing tasks. The developer need only be aware of the potentially re-entrant nature of the operation processing routines to write them successfully for this environment. The basic design and control flow within an operation processing routine need not be changed to operate concurrently, however.
2. **Uniform Dispatching to Operation Processing Code**: The main body of an object manager receives requests, determines which operation is being invoked, and dispatches to the appropriate operation processing routine. The manager development tools generate the operation dispatcher for a manager, including use of the tasking package to allow concurrent operation processing.
3. **Support for Heterogeneous Implementations**: Operation parameters are automatically translated to and from the Cronus canonical data representations provided by the Message Structure Library (MSL). The developer need only be concerned with the native internal forms of data; the manager development tools take care of any conversions necessary for transmitting data among heterogeneous Cronus implementations.
4. **Management of Stored Object Descriptors**: Nearly every type of object requires some non-volatile storage to retain the object's descriptor. A package of routines for maintaining the object descriptor is provided by the manager development tools.
5. **Access Control**: All operations are automatically checked for required access permissions before they are allowed to be carried out, and no operation is allowed to proceed without required access rights.

6. **Multiple Managers Per Process**: Multiple object types may be managed by a single manager process transparently; the dispatcher automatically routes requests to the appropriate operation processing routines. Combining the support of different object types within a single manager can result in improved performance, through techniques such as code and data sharing.

7. **Operation Processing Routines for Common Operations**: The manager development system provides a library of processing routines for operations inherited from types higher in the type hierarchy. These standard operations need not be reimplemented by object managers, since they are not dependent on type-specific information. Included in the set of standard operations which apply to all Cronus objects are operations for creating, removing, and locating objects, and operations for integration with the Cronus Access Control and Monitoring and Control systems. This library of routines can often supply most of the operations that a type supports, and only a few new operation processing routines need to be written.

8. **Client Interface Library for New Object Types**: The manager development software automatically generates interface subroutines that format operation invocation messages, invoke the operations, and collect the results. These interface subroutines provide Cronus client applications with a RPC-style interface to Cronus operations.

9. **Interactive Operation Invocation**: Operations defined in the type definition database can be invoked directly by a user through interactive programs called **auth** and **ui**. These programs automatically acquire the appropriate operation interface descriptions needed for invoking operations on particular object types. These programs can be used directly by the manager developer for debugging, and can also be used to support a user-level command when invocation of a single operation maps into such a command.

10. **Integrated Documentation Maintenance**: A special annotation feature of the object specification language provides a mechanism incorporating documentation describing the operation interface and associated canonical types. Another program retrieves this information to generate typeset manual articles for User's Manuals.

Each new object type is described using a non-procedural definition language called **Conduit**. A special purpose object manager responsible for the *type definition database* interprets this language, and stores object type descriptions in a database. Each object type definition is itself a Cronus object. Once an object type description is stored in the database, this manager can generate program code which implements large parts of the application object manager automatically. This generated code when compiled and linked with a collection of standard library routines and user supplied operation processing routines, comprises a complete production version of the application object manager. In addition to the object manager, the automatic code generator produces an operation interface for client programs.

13.2. Object Type Definition

Designing a distributed application for Cronus consists of choosing object types and operations, and detailing the interactions among client programs and objects, and between the objects themselves. Once the overall design of the application has been completed, detailed design of the individual object types and the operations that they respond to can begin. The application developer specifies the operation protocol details of a new application object type using **Conduit**. A user program sends this definition to the type definition manager, where the new object type object is created and stored in the type definition database maintained by the manager. A second user program and simple implementation definition instruct the type definition manager to automatically generate code to implement most of the object manager for the new type, as well as a client interface subroutine library, and optionally, documentation for the new object type.

13.2.1. The Conduit Language

When a developer specifies a Cronus type using **Conduit**, he is specifying the behavior and implementation of a new class of Cronus objects. The Cronus object model provides a mechanism for a type to inherit characteristics from another type similar but less specific in its special properties. All Cronus types are subtypes of some other type, from which they inherit characteristics. The inheritance relationships among Cronus types define a *type hierarchy*. At the top of the type hierarchy is one type, **CT Object**, that is not a subtype of any other type. This type defines characteristics that all objects share.

Conduit provides for the inheritance of type definitions in support of the Cronus object model. This means that only the portions of a type definition that are specific to the type being defined must be included, and all other portions of the type definition may be inherited. Most sections of a type definition are optional, since it is possible to inherit all the information for a section of the type definition.

A **Conduit** definition consists of several sections, which appear in a fixed order. The first section includes information such as the type's position in the type hierarchy and the names of access rights that apply to the type as a whole. Subsequent sections define data formats, parameter labels, error codes, and operation parameters and access rights. Because the operations defined on the **generic object** for a type may be different than those defined on the specific objects of the type, operations and access rights are separately specified for generic and specific objects.

13.2.2. Elements of a Type Definition

An input file contains one or more type definitions, where each type definition consists of five sections: the **type** declaration, the **canonical type** section, the **error** section, the **key** section, and the **operation** section. Each section is composed of individual declarations of canonical types, errors, keys, or operations. A semicolon is used at the end of each declaration to terminate it, and commas are used between clauses of declarations as separators. Only the type declaration is required in a type definition; all other sections are optional if the sections' declarations are inherited from a type's supertype.

The complete syntax description for **Conduit** follows, to illustrate the kinds of definition capabilities that the language has. The **Cronus User's Manual**, section 4, has a complete description of the language and its use.

Syntax:

type *name* , *number* ,
    abbrev is <*string*>
    |**subtype of** <*type-name*>|
    |**rights are** <*name*> |= <*bit-number*>|, ...|
    |**generic rights are** <*name*> |= <*bit-number*>|, ...|
    |**is primal**|
    |**is** |**fully**| **replicated**|
    |**has no instances**|
    |**annote** <*string*>|;

|**variable**| **cantype** <*name*> |= <*number*>|
        representation is <*string*> |: **record**
            <*name*>: |**array of**| <*cantype-name*>,
            ...
        end <*string*> |
        representation is <*string*>
            |{ <*name*> |= <*number*>|, ... }|
        |**annote** <*string*>|
            ...

key <*name*> |= <*number*>|: |**array of**| <*cantype-name*>
    |**annote** <*string*>|;
            ...

error <*name*> |= <*number*>|
    annote (<*string*>);
            ...

|**generic**| **operation** <*name*> |= <*number*>| (|<*parameter*>, <*parameter*>, ...)
        |**returns** (<*parameter*>, <*parameter*>, ...)|
        |**requires** <*right-name*>, <*right-name*>, ... |
        |**annote** <*string*>|;

|**optional**| <*key-name*>: |**array of**| <*cantype-name*>
        |**annote** <*string*>|
            ...

end type <*type-name*>;

13.2.3. Conduit Processor Implementation

After writing a specification for a new object type, the programmer uses a Cronus command to enter the new type definition into the *protocol database*. The command invokes an operation on the type definition manager, which manages this database. The **Conduit** source code is sent unedited to the type definition manager in a Cronus operation message, usually using the large message facility of Cronus IPC. The type definition manager then analyzes the new type definition using a language parser constructed with the standard UNIX compiler generation tool, **yacc**. If there are errors in the syntax or semantics of the type definition, these are indicated in the reply message to the invoking command.

After parsing the specification and converting it to an intermediate representation suitable for storing in the protocol database, the manager enters the new type definition into the database and replies with a success completion code to the command. Type definitions are full-fledged Cronus objects, including all operations (ie. access control, etc) inherited from the parent CT_Object type. There are a number of operations defined for type definition objects, and the application development tools access type definition objects using standard Cronus techniques. Storing type definitions as objects has a number of advantages including, making them globally accessable, access controlled, and replicated for reliability.

The protocol database itself is a standard object database, and type definitions are stored as large canonical types in non-volatile storage. Each type definition object contains a link to its parent object type in the type hierarchy, implementing type inheritance. All canonical type definitions, keys, errors, and operations defined for a given type definition object are stored with that object in the object database of the type definition manager.

13.2.4. Generating Application Code Automatically

The **Genmgr** *command processes a non-procedural description of object manager implementation* details, by sending this description to the type definition manager in much the same way as **Conduit** definitions are processed. Based on this description and the information already stored in the protocol database by **Conduit**, **Genmgr** generates source code for the common parts of the manager, such as message parsing, dispatching, access control, etc. The generated source code is then compiled, and linked with both the user-written operation processing routines for handling operations specific to the Cronus type, and the manager run-time libraries containing operation processing routines for operations shared among a number of managers. The resulting executable image is the object manager for the new type.

The source code generated by **Genmgr** is portable to any system supporting Cronus and the C programming language. To build the object manager for a host architecture which does not yet support the manager, the programmer compiles the **Genmgr** output and the user-written processing routines using a compiler for that host architecture, and links them with libraries available for that type of host.

The applications programmer is required to write the **Conduit** type description, the **Genmgr** implementation description, and the operation processing routines for operations specific to the type being defined. The development tools do the rest of the work, supplying much of the code for the manager, customized to work with the user-supplied portions. In addition to components of the object manager for the new type, the **Genmgr** program also produces an interface library used by applications to invoke operations on objects of the new type.

13.3. Components of an Object Manager

An object manager consists of a framework of systems software providing the control structures and standard capabilities of managers, and user-written operation processing routines called by this control structure to carry out the actual work of the manager. There are three types of systems software code automatically generated by the application development tools. There are the underlying support, manager control routines, and standard object facilities.

13.3.1. The Tasking Package

Object managers must be capable of handling multiple requests simultaneously. If an object manager could only handle a single request at a time, requests might be queued for long periods of time awaiting the sequential processing of previous operations, even if such processing involved idle time while suboperations completed. Performance would be seriously degraded, because managers would not be making the best use of available computing resources.

Unfortunately, it has been our experience that the asynchronous independent processes with virtual memory and preemptive scheduling offered by traditional operating systems is too expensive in its implementation to be of use in this instance. What is needed is a 'lightweight process' mechanism, which provides very simple asynchronous processing with as little performance penalty as is possible. Such a mechanism dispenses with independent virtual address spaces, preemptive scheduling, and a separation between user and system code and data.

The Cronus Tasking Package is a portable subroutine library which implements separate *tasks*, independent threads of control within the same address space. Tasks may be created, suspended, resumed, signalled, and destroyed. This asynchronous processing technique is at the foundation of our object managers.

13.3.2. Work-In-Progress Lists

An object manager is a single process to the local operating system. IPC messages are queued for the manager process as a whole, and replies to messages invoked by tasks within the manager must be dispatched to the appropriate tasks. The *work-in-progress* list is an abstract data structure used to store arbitrary task contexts, which are awaiting receipt of a reply message. The appropriate task context will be restored and the task run when a reply is received by the manager.

13.3.3. Object Manager Control Flow

The control flow of an object manager is mediated entirely by the tasking package. The manager consists of a main routine which initializes the tasking package and starts the three tasks which together control the activity of the manager; the initialization, receive, and idle tasks. The main routine of an object manager performs some global initialization, creates the three main tasks, and starts the tasking package, relinquishing control to a non-preemptive, round-robin scheduler.

13.3.3.1. The Initialization Task

The *initialization task* is responsible for performing the type-specific initializations required for each type managed by the object manager. These initializations are performed by user-written routines. The initialization task calls each of these routines in turn. Type-specific initializations might include consistency checks or crash recovery processing, set-up of initial processing conditions such as logging levels, and synchronization of replicated objects with other copies of the objects stored elsewhere in Cronus.

Because manager initialization is performed after the tasking package has been granted control, initialization may consist of any type of processing, including invocation of operations on other objects in the system.

13.3.3.2. The Receive Task

The *receive task* initiates and controls the scheduling of most of the activity of the manager by dispatching incoming invocation and reply messages to tasks which process them asynchronously. This task uses tables generated by the application development tools to process request messages, and the Work-In-Progress lists to process replies from suboperations invoked by other tasks within the manager.

A new task is created by the receive task when a request message is received. This task then converts the message itself from canonical to internal form, performs an access control check, retrieves the requested object's instance variables from the object database, and then calls the appropriate user-written operation processing routine to actually perform the operation. Any of these steps, including the operation processing routine itself may invoke operations on other objects. When the subtask has invoked an operation and is ready to wait for the reply, it calls a version of the Cronus **ReceiveReply** library routine. This routine creates a Work-In-Progress entry for the task, including all task context which needs to be saved for subsequent processing of the reply. This entry is entered into the Work-In-Progress list, and then the task relinquishes control to the task scheduler. When a reply message is received by the receive task, it looks up the operation identifier for the reply in the Work-In-Progress list, places the received message in a buffer supplied as part of the Work-In-Progress entry, and unblocks the task which is waiting for this reply.

13.3.3.3. The Idle Task

The *idle task* intervenes in the normal round-robin scheduling of the tasking package to implement priority processing. Because the idle task actually runs at a higher priority than any other task, it is guaranteed to get control after every task switch within the manager. It checks to be sure that the task being resumed by the receive task is in fact the highest priority task ready to run. If it is, this task is resumed, otherwise higher priority tasks are resumed first. Priority is determined by a parameter of the process bindings of the process which initially invoked an operation.

13.3.4. Standard Operation Processing Routines

Operation processing routines for the operations which a manager inherits from type CT_Object are contained in a subroutine library. These routines, perform a large number of useful operations, including:

- responding to object location requests
- maintaining access control parameters for the object
- setting and querying user and system parameters
- implementing generic monitoring and control operations
- providing for type-independent backup, restore, replication, and migration of objects
- implementing dynamic type description operations

Many object types are implemented almost entirely from these supplied operation processing routines, and require only a few additional operations to implement their entire function.

13.4. Client Program Interface

In addition to the object manager, the application development tools also automatically generate a subroutine library providing a client interface for new operations defined as part of an application's object types. These routines provide an interface which resembles a *remote procedure call* for each operation. The client program passes operation parameters to the library routine, which constructs the request message, invokes the operation, receives and parses the reply, and returns reply data and status using familiar programming techniques. Client program developers may use these interface routines just as they would use any standard run-time library. The distributed nature of the processing is effectively hidden behind the subroutine interface to these routines.

13.5. Other Support Features

13.5.1. Documentation Generation

As part of the **Conduit** specification for a new object type, the application developer may include annotations for most of the definition clauses. A documentation generation tool then takes these annotations, together with the overall structure and definition of the object type, and generates a command file targetted to the **troff** typesetter language available on **UNIX** systems. This command file produces a typeset article suitable for inclusion in the **Cronus User's Manual**. Other typesetting languages and formats could be easily supported as well.

13.5.2. Table-Driven User Interface Programs

The application development tools include two 'universal' user interface programs capable of constructing request messages for any operation known to the type definition manager's protocol database. These two programs, called **auth** and **ui**, can be used by application developers for testing and evaluating new application object managers. They can also be used for building simple commands to invoke operations, using the local operating system's command interpreter to run command scripts that call them.

14. Advanced Development Model Hardware

The Advanced Development model of the Cronus distributed operating system currently has access to several large mainframe computers, and has exclusive access to several minicomputers, workstations, GCEs, and a gateway. The minicomputers are a mixture of Digital Equipment Corporation VAX 11/750 and BBN C70s computers; the workstations are SUN systems; the GCEs are Multibus computers with M68000 central processors; and the gateway is a DEC LSI-11 based computer.

The mainframe systems are used for development support and peripheral device support. The systems are mainly VAX 11/780 and 11/785 systems which provide timesharing support to the division at BBN. These hosts also run Cronus, concurrent with the timesharing load, to support access to files, disks and other and peripheral devices.

The VAX 11/750, to which we have exclusive access, provides a VMS-based software development environment. Its purpose in the ADM is to provide a limited integration host. Since it is a large well-supported system, it contains its own development environment, and we also use it as a source of computer power for general tasks, both to off-load the other systems and to test real usage of the Cronus heterogeneous host environment. The VAX is configured to reflect its usage as a software development machine.

The C70 computers are configured as general development machines. The first, C70-1, is the site of the majority of the development work since it supports both the C70 development tools and those of the GCEs. We will rent time on a second C70, C70-2, which will be used to exercise Cronus support for reliable redundant hosts, and to test scalability. Both C70s will run UNIX version 7 as released by BBN Computer Corporation and modified by the Cronus project.

The SUN workstations are each configured with at least 2 Mbytes of memory and 120 Mbytes of disk. Both systems run UNIX and support a window oriented user interface. Some systems also supports color monitors.

The Cronus system has several GCEs, configured for a variety of tasks. Their configurations will vary over time, as we perform different experiments on the network, and as we make board substitutions to make one GCE perform functions of another which is temporarily out of service. The configuration table for the GCEs should be regarded as only a typical set of GCE configurations.

The Cronus gateway is implemented on an DEC LSI-11 computer. This would normally be a task for a GCE; however, standard internet gateways are currently implemented on LSI-11, and adoption of the LSI-11 gateway allows us to obtain an off-the-shelf implementation. The next generation of internet gateways is expected to be built on M68000 computers, and at that time we will probably move the gateway to a GCE.

VAX 11-785	12 Mbytes main memory 1773 Mbytes of disk 1600/6250 BPI tape drive Ethernet Interface Berkeley Unix 4.2 Operating System
VAX 11-780	6 Mbytes main memory 811 Mbytes of disk Ethernet Interface Berkeley Unix 4.2 Operating System
VAX 11-750	1 Mbytes main memory 1 160 Mbyte Winchester disk Magnetic tape drive, 1600 bpi, 40 ips MDI high speed synchronous serial interface 3COM Ethernet Interface VMS Operating System
μVAX-II	5 Mbytes main memory 1 380 Mbyte Winchester disk Ethernet Interface
C70-1	1 Mbytes main storage 2 80 Mbyte removable disk drives Magnetic Tape Drive, 800/1600 bpi, 125 ips (Cipher) Arpanet 1822 LHDH interface Ethernet interface (using Interlan protocol module)
C70-2	1/2 Mbytes main storage 2 160 Mbyte removable disk drives Arpanet 1822 LHDH interface Ethernet interface (using Interlan protocol module)

Software Development Hosts
Table 14.1

SUN 100	2 Mbytes main storage 1 80 Mbyte Winchester disk 15" b/w BitMap display UNIX operating system
SUN 120	2 Mbytes main storage 1 120 Mbyte Winchester disk 19" b/w BitMap display UNIX operating system
SUN 120	2 Mbytes main storage 1 130 Mbyte Winchester disk 19" b/w BitMap display 19" color BitMap display UNIX operating system
SUN 3/160	4 Mbytes main storage 1 380 Mbyte Winchester disk 19" high resolution color BitMap display UNIX operating system

Workstations
Table 14.2

MassComp	M68010 processor with 1Mbyte main memory 168 Mbyte Winchester disk Ethernet Interface
Forward Technology	M68000 processor with 256 Kbytes memory Micro-Memory 256 Kbyte memory board 8-line RS-232 serial interface 3COM Ethernet Interface 8-slot Multibus backplane
Forward Technology	M68000 processor with 256 Kbytes memory Micro-Memory 256 Kbyte memory board 8-line RS-232 serial interface 3COM Ethernet Interface 8-slot Multibus backplane

Generic Computing Elements -- Typical Configurations
Table 14.3

Gateway	LSI11/03 processor card 64 Kbyte memory card DLV11J 4 line terminal card MRV11C ROM card (bootstrap) ACC 1822 interface with DMA Interlan NI2010 QBUS Ethernet controller BBN FNV11 Fibernet interface MDB backplane and power-supply.

Gateway Configuration
Table 14.4

15. Virtual Local Network

15.1. Purpose and Scope

The Cronus Virtual Local Network (VLN) provides interhost message transport in the Cronus Distributed Operating System. The VLN client interface is available on every Cronus host. Client processes can send and receive messages using specific, broadcast, or multicast addressing.

The VLN stands in place of a direct interface to the physical local network (PLN). This additional level of abstraction is defined to meet two major system objectives:

- *Compatibility.* The VLN is compatible with the Internet Protocol (IP) and with higher-level protocols, such as the Transmission Control Protocol (TCP), based on IP.
- *Substitutability.* Cronus software built above the VLN is dependent only upon the VLN interface and not its implementation. It is possible to substitute one physical local network for another provided that the VLN interface specification is satisfied.

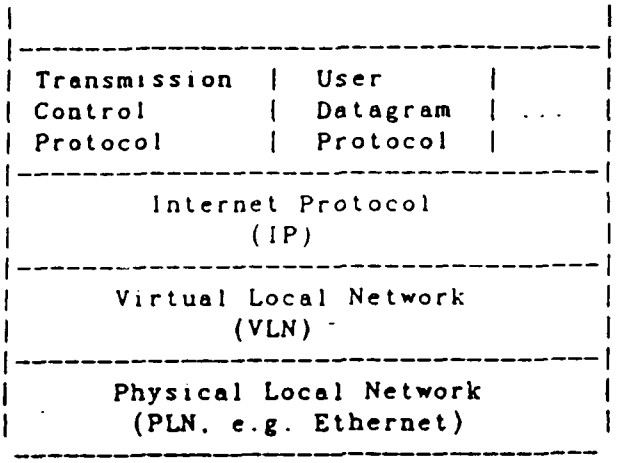
This description assumes the reader is familiar with the concepts and terminology of the DARPA Internet Program; reference [NIC 1982] is a compilation of the important protocol specifications and other documents. Documents in [NIC 1982] of special significance here are [Postel 1981a] and [Postel 1981b].

The Advanced Development Model ADM will be connected to the ARPANET, and it is important that the ADM conform to the standard and conventions of the DARPA internet community. In addition, a large body of software has evolved, and continues to evolve, in the internet community. For example, protocol compatibility permits Cronus to assimilate existing software components providing electronic mail, remote terminal access, and file transfer.

The substitutability goal reflects the belief that different instances of Cronus will use different physical local networks. Substitution may be desirable for reasons of cost, performance, or other properties of the physical local network such as mechanical and electrical ruggedness.

Figure 1 shows the position of the VLN in the lowest layers of the Cronus protocol hierarchy. The VLN interface specification leaves programming details of the interface and host-dependent issues unspecified. The precise representation of the VLN data structures and operations will vary from machine to machine, but the functional capabilities of the interface are the same regardless of the host.

The VLN is completely compatible with the Internet Protocol as defined in [Postel 1981b]. No changes or extensions to IP are required to implement IP above the VLN.

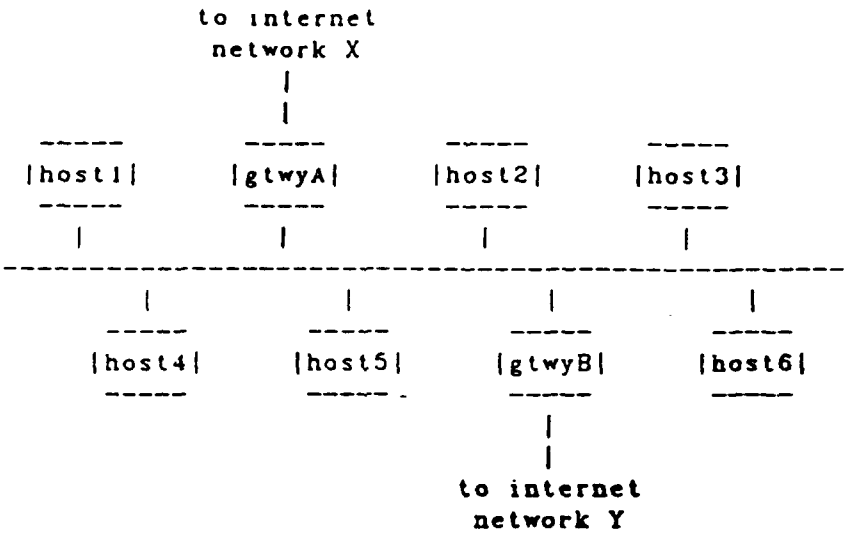


Cronus Protocol Layering
Figure 15.1

15.2. The VLN-to-Client Interface

The VLN layer provides a datagram transport service among hosts in a Cronus *cluster*, and between these hosts and other hosts in the DARPA internet. The hosts belonging to a cluster are attached to the same physical local network. Communication with hosts outside the cluster is achieved through *internet gateways*, shown in Figure 2, connected to the cluster. The VLN routes datagrams to a gateway if they are addressed to hosts outside the cluster, and delivers incoming datagrams to the appropriate VLN host. A VLN is a network in the internet, and thus has an internet network number[19].

[19]The network numbers for the PLN and VLN may be the same or different. If the numbers are different, the gateways are somewhat more complex. Either approach is consistent with the internet model.



A Virtual Local Network Cluster
Figure 15.2

The VLN interface will have one client process on each host, normally the host's IP implementation. The VLN performs no multiplexing/ demultiplexing function.

The structure of messages which pass through the VLN is identical to the structure of internet datagrams. The VLN definition assumes that there is a well-defined representation for internet datagrams on any host supporting the VLN interface. The argument name "Datagram" in the VLN operation definitions below refers to this well-defined but host-dependent datagram representation.

The VLN guarantees that a datagram of 576 or fewer octets can be transferred between any two VLN clients. Although larger datagrams may be transferred between some client pairs, clients should avoid sending datagrams exceeding 576 octets unless there is clear need to do so. The sender must be certain that all hosts involved can process the oversized datagrams.

The internal representation of an VLN datagram is not included in the specification, and may be chosen for implementation convenience or efficiency.

Although the structure of internet and VLN datagrams is identical, the VLN-to-client interface places its own interpretation on internet header fields, and differs from the IP-to-client interface in significant respects:

1. The VLN layer uses only the Source Address, Destination Address, Total Length, and Header Checksum fields in the internet datagram; other fields are accurately transmitted from the sending to the receiving client.
2. Internet datagram fragmentation and reassembly is not performed in the VLN layer, nor does the VLN layer implement any aspect of internet datagram option processing.
3. At the VLN interface, a special interpretation is placed upon the Destination Address in the internet header, which allows VLN broadcast and multicast addresses to be encoded in the internet address structure.
4. With high probability, duplicate delivery of datagrams sent between hosts on the same VLN does not occur.
5. Between two VLN clients S and R in the same Cronus cluster, the sequence of datagrams received by R is a subsequence of the sequence sent by S to R; a stronger sequencing property holds for broadcast and multicast addressing.

In the DARPA internet, an *internet address* is defined to be a 32-bit quantity that is partitioned into two fields, a *network number* and a *local address*. VLN addresses share this basic structure, but it attaches special meaning to the local address field of a VLN address.

Each network is assigned a *class* (A, B, or C), and a network number. The partitioning of the 32-bit internet address into network number and local address fields as a function of the class of the network is shown in Table 15.1.

	Width of Network Number	Width of Local Address
Class A	7 bits	24 bits
Class B	14 bits	16 bits
Class C	21 bits	8 bits

Internet Address Formats
Table 15.1

The bits not included in the network number or local address fields encode the network class, e.g., a 3 bit prefix of 110 designates a class C address (see [Postel 1981a]).

The interpretation of the local address field is the responsibility of the network. For example, in the ARPANET the local address refers to a specific physical host. VLN addresses, in contrast, may refer to all hosts (broadcast) or groups of hosts (multicast) in a Cronus cluster, as well as specific hosts inside or outside of the cluster. Specific, broadcast, and multicast addresses are all encoded in the VLN local address field [20]. The meaning of the local address field of a VLN address is defined in Table 15.2.

Address Modes	VLN Local Address Values
Specific Host	0 to 1,023
Multicast	1,024 to 65,534
Broadcast	65,535

VLN Local Address Modes
Table 15.2

In order to represent the full range of specific, broadcast, and multicast addresses in the local address field, a VLN network should be either class A or class B.

The VLN does not attempt to guarantee reliable delivery of datagrams, nor does it provide negative acknowledgements of damaged or discarded datagrams. It does guarantee that received datagrams are accurate representations of transmitted datagrams.

The VLN guarantees that datagrams will not replicate during transmission, so each intended receiver, a given datagram given to the VLN by higher levels is received once or not at all[21].

Between two VLN clients S and R in the same cluster, the sequence of datagrams received by R is a subsequence of the sequence sent by S to R, that is datagrams are received in order, possibly with omissions. A stronger sequencing property holds for broadcast and multicast transmissions. If receivers R1 and R2 both receive broadcast or multicast datagrams D1 and D2, either they both receive D1 before D2, or they both receive D2 before D1.

While a VLN could be implemented on a long-haul or virtual-circuit-oriented PLN, these networks are generally ill-suited to the task. The ARPANET, for example, does not support broadcast or multicast addressing modes, nor does it provide the VLN sequencing guarantees. If the ARPANET were the base for a VLN implementation, broadcast and multicast would have to be constructed from specific addressing, and a network-wide synchronization mechanism would be required to implement the guarantees. Although the compatibility and substitutability benefits might still be achieved, the

[20] The ability of hosts outside a Cronus cluster to transmit datagrams with VLN broadcast or multicast destination addresses into the cluster may be restricted by the cluster gateway(s), for reasons of system security.
[21] A protocol operating above the VLN layer (e.g., TCP) may employ a retransmission strategy; the VLN layer does nothing to filter duplicates arising in this way.

implementation would be costly, and performance poor.

A good implementation base for a Cronus VLN would be a high-bandwidth local network with all or most of these characteristics:

1. The ability to encapsulate a VLN datagram in a single PLN datagram.
2. An efficient broadcast addressing mode.
3. Natural resistance to datagram replication during transmission.
4. Sequencing guarantees like those of the VLN interface.
5. A strong error-detecting code (datagram checksum).

Good candidates include Ethernet, the Flexible Intraconnect, and Pronet, among others.

15.3. A VLN Implementation Based on Ethernet

The Ethernet local network specification is the result of a collaborative effort by Digital Equipment Corp., Intel Corp., and Xerox Corp. The Version 1.0 specification [DEC 1980] was released in September 1980. Useful background information on the Ethernet internet model is supplied in [Dalal 1981].

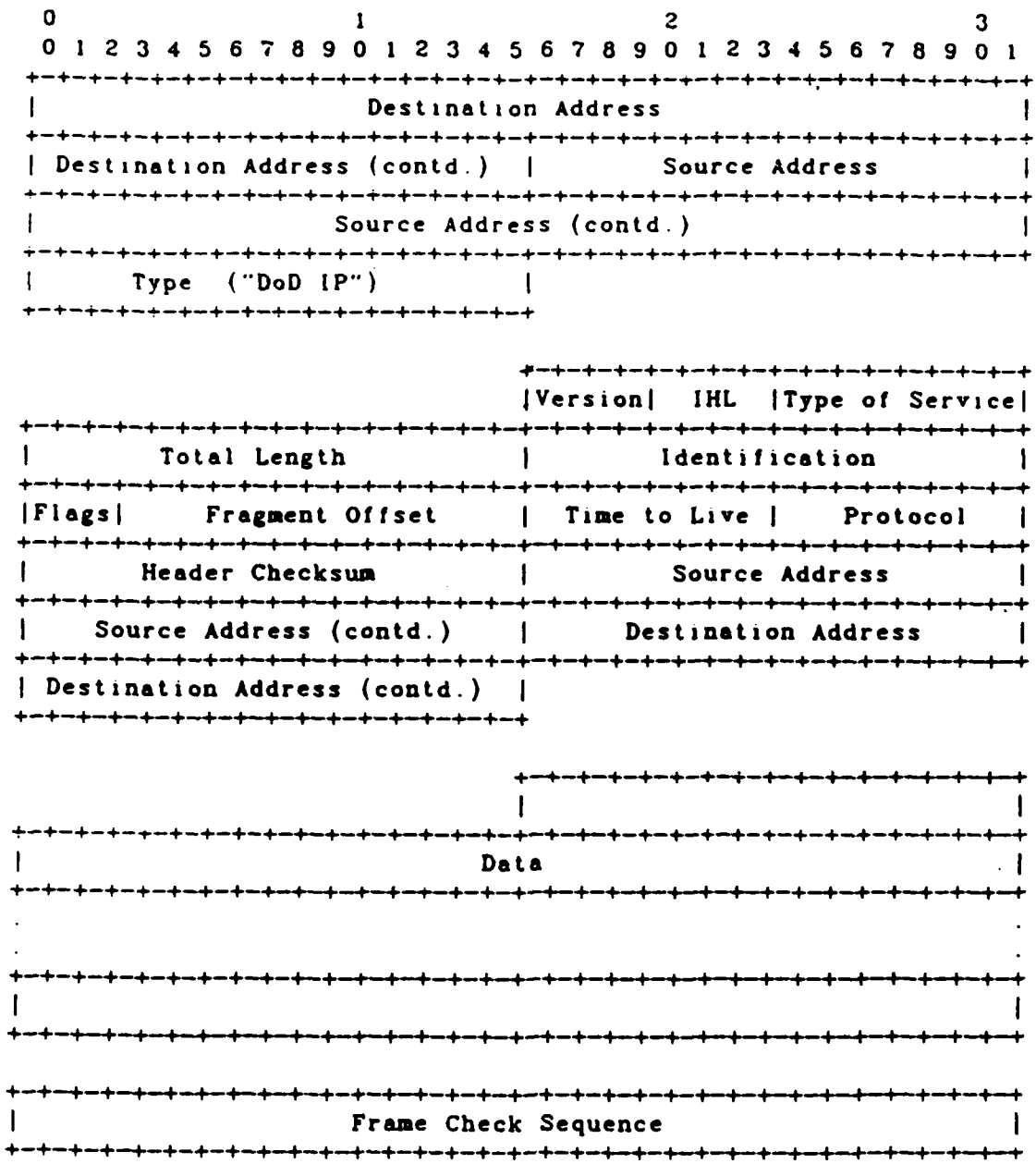
The addresses of specific Ethernet hosts are arbitrary 48-bit quantities, not under the control of the DOS. The VLN implementation must map VLN addresses to specific Ethernet addresses. The mapping can not be maintained manually in each VLN host, because manual procedures are too cumbersome and error-prone for a local network with many hosts, each of which may join and leave the network frequently. A protocol is described below which allows a host to construct the mapping dynamically, beginning only with knowledge of its own VLN and Ethernet host addresses.

An internet datagram is encapsulated in an Ethernet frame by placing the internet datagram in the Ethernet frame data field, and setting the Ethernet type field to "DoD IP", as shown in Figure 15.3.

The Ethernet octet ordering is required to be consistent with the IP octet ordering. If IP(i) and IP(j) are internet datagram octets and $i<j$, and EF(k) and EF(l) are the Ethernet frame octets which represent IP(i) and IP(j) once encapsulated, then $k<l$. Bit orderings within octets must also be consistent.

Each VLN component maintains a virtual-to-physical address map (the VPMap) which translates a 32-bit specific VLN host address to a 48-bit Ethernet address. The VPMap data structure and the operations on it will implemented using hashing techniques.

Each host controller has an Ethernet host address (EHA) to which it responds. The EHA is determined by Xerox and the controller manufacturer. In addition, the VLN assigns a multicast-host address (MHA) to each host. This multicast address is constructed from the local host portion of the internet address.



An Encapsulated Internet Datagram
Table 15.3

When the VLN client sends a datagram to a specific host, the local VLN component encapsulates it and transmits it without delay. The Source Address in the Ethernet frame is the EHA of the sending host. The Ethernet Destination Address is formed from the destination VLN address in the datagram, and is either:

- the EHA of the destination host, if the sending host knows it, or
- the MHA formed from the host number in the destination VLN address, as described above, if the sending host does not know the EHA corresponding to the host number.

When a VLN component receives an Ethernet frame with type "DoD IP", it decapsulates the internet datagram and delivers it to its client. If the frame was addressed to the EHA of the receiving host, no further action is taken. If the frame was addressed to the MHA of the receiving host, the VLN component broadcasts an update for the VPMaps of the other hosts. The other hosts can then use the EHA of this host for future traffic. If the MHA is represented as a sequence of octets in hexadecimal, it has the form:

A B C D E F

09-00-08-00-hh-hh

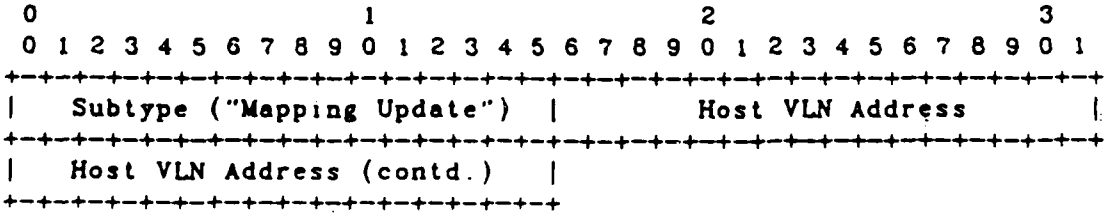
A is the first octet transmitted, and F the last. The two octets E and F contain the host local address:

E F

000000hh hhhhhhhh

MSB LSB

The type field of the Ethernet frame containing the update is "Cronus VLN", and the format of the data octets in the frame is:



When a local VLN component receives an Ethernet frame with type "Cronus VLN" and subtype "Mapping Update", it performs a StoreVPPair operation using the Ethernet Source Address field and the

host VLN address sent as frame data.

A VLN datagram will be transmitted in broadcast mode if the specifies the VLN broadcast address (local address = 65,535, decimal) as the destination. The receiving VLN component merely decapsulates and delivers the VLN datagram.

The implementation of multicast addressing is more complex. Each host defines the number of multicast addresses which can be simultaneously "attended" (listened to). This number is a function of the particular Ethernet controller hardware and of the resources that the host dedicates to multicast processing. The VLN protocol permits a host to attend any number of multicast addresses, from 0 to 64,511 (the entire VLN multicast address space), independent of the controller in use.

It is possible to implement the VLN multicast mode using only the Ethernet broadcast mechanism. Every VLN host would receive and process every VLN multicast, discarding uninteresting datagrams. More efficient operation is possible if some Ethernet multicast addresses are used, and if the Ethernet controller has multicast recognition which automatically discard misaddressed frames.

There is no standard for multicast recognition. The 3COM Model 3C400 controller performs no multicast address recognition. It passes all multicast frames to the host for further processing. The Intel Model iSBC 550 controller permits the host to register a maximum of 8 multicast addresses with the controller, and the Interlan Model NM10 controller permits a maximum of 63 registered addresses.

A VLN-wide constant, Multicast_Registered, is equal to the smallest number of Ethernet multicast addresses that can be simultaneously attended by all hosts in the VLN. A network composed of hosts with the Intel and Interlan controllers mentioned above, for example, would have Multicast_Registered equal to 7 [22]; a network composed only of hosts with 3COM Model 3C400 controllers would have Multicast_Registered equal to 64,511, since the controller itself does not restrict the number of Ethernet multicast addresses to which a host may attend [23].

A mapping is defined which translates the VLN multicast address to an Ethernet multicast address. The first Multicast_Registered VLN multicast addresses are assumed to be attended by each host. The local address portion of the internet address of a VLN multicast channel is a decimal integer M in the range 1,024 to 65,534.

1. (M - 1,023) <= Multicast_Registered. In this case, the Ethernet multicast address is

   09-00-08-00-mm-mm

2. (M - 1,023) > Multicast_Registered. The Ethernet broadcast address is used. A VLN component which attends VLN multicast addresses in this range must receive all broadcast frames, and select those with VLN destination address corresponding to the attended multicast

[22] Multi_Registered is 7, rather than 8, because one multicast slot in the controller is reserved for the host's MHA.
[23] For the Cronus Advanced Development Model, Multicast_Registered is currently defined to be 60.

address.

Delivered datagrams are accurate copies of transmitted datagrams because VLN components do not deliver datagrams with invalid Frame Check Sequences. A 32-bit CRC error-detecting code is applied to Ethernet frames.

Datagram duplication does not occur because the VLN layer does not perform retransmissions, the primary source of duplicates in other networks. Ethernet controllers do perform retransmission as a result of collisions on the channel, but the collision enforcement mechanism or "jam" assures that no controller receives a valid frame if a collision occurs.

The sequencing guarantees hold because mutually exclusive access to the transmission medium defines a total ordering on Ethernet transmissions, and because a VLN component buffers all datagrams in FIFO order.

15.4. VLN Operations

There are seven functions defined at the VLN interface. An implementation of the VLN interface has wide latitude in the presentation of these operations to the client; for example, the functions may or may not return error codes.

The functions are to occur synchronously or asynchronously with respect to the client's computation. We expect that the ResetVLNInterface, MyVLNAddress, SendVLNDatagram, PurgeMAddresses, AttendMAddress, and IgnoreMAddress operations will be synchronous with respect to the client. ReceiveVLNDatagram will usually be asynchronous; that is, the client initiates the operation, continues to compute, and at some later time is notified that a datagram is available.

ResetVLNInterface()

> The VLN for this host is reset. For the Ethernet implementation, the operation ClearVPMap is performed, and a frame of type "Cronus VLN" and subtype "Mapping Update" is broadcast. This operation does not affect the set of attended VLN multicast addresses.

MyVLNAddress()

> Returns the VLN address of this host.

SendVLNDatagram(Datagram)

> When this operation completes, the VLN layer has copied the Datagram. The transmitting process cannot assume that the message has been delivered when SendVLNDatagram completes.

ReceiveVLNDatagram(Datagram)

When this operation completes, Datagram is a representation of a VLN datagram which has not previously received.

PurgeMAddresses()

When this operation completes, no VLN multicast addresses are registered with the local VLN component.

AttendMAddress(MAddress)

If this operation returns True then MAddress, which must be a VLN multicast address, is registered as an alias for this host, and messages addressed to MAddress by VLN clients will be delivered to the client on this host.

IgnoreMAddress(MAddress)

When this operation completes, MAddress is not registered as a multicast address for the client on this host.

Whenever a Cronus host comes up, ResetVLNInterface and PurgeMAddresses are performed on the VLN. A VLN component may depend upon state information obtained dynamically from other hosts, and there is a possibility that incorrect information might enter a component's state tables. A cautious VLN client could call ResetVLNInterface periodically to force the VLN component to reconstruct the tables.

A VLN component will limit the number of multicast addresses to which it will simultaneously attend; if the client attempts to register more addresses than this, AttendMAddress will return False with no other effect.

The VLN layer does not guarantee buffering for datagrams at either the sending or receiving host(s). It does guarantee that a SendVLNDatagram function performed by a VLN client will eventually complete: this implies that datagrams may be lost if buffering is insufficient and receiving clients are too slow.

16. Broadcast Repeater

This section presents the problem of multi-network broadcasting and our motivation for solving this problem. We discuss different solutions to extending a broadcast domain and why we chose the one that has been implemented. In addition, there is information on the implementation itself and some notes on its performance.

16.1. The Problem

Communication in Cronus is built upon the TCP and UDP protocols. The broadcast facilities offered by the Local Area Network (**LAN**) are used for dynamically locating managers and resources on other hosts and collecting status information from a collection of managers. However, broadcasts are not available when the clients of one LAN wish to access resources of another LAN using the DARPA Internet: broadcasted packets are only received by hosts on the physical network on which the packet was broadcast. As a result, if no additional support is provided clients can only use resources connected to the client's LAN.

Since the range of a Cronus cluster is not intended to be limited to the boundaries of a single LAN, we have extended our broadcasting domain to include hosts on distant LANs in order to experiment with clusters that span several physical networks. Cronus predominantly uses broadcasting to communicate with a subset of the hosts that actually receive the broadcasted message. A multicast mechanism would be more appropriate, but is unavailable in our network implementations, so we chose broadcast for the initial implementation of Cronus utilities.

16.2. Our Solution

The technique we implemented to experiment with the multi-network broadcasting problem can be described as a *broadcast repeater*. A broadcast repeater is a mechanism which transparently relays broadcast packets from one LAN to another, and may also forward broadcast packets to hosts on a network which doesn't support broadcasting at the link-level. This mechanism provides flexibility while still taking advantage of the convenience of LAN broadcasts.

Our broadcast repeater is a process on a network host which listens for broadcast packets. These packets are picked up and retransmitted, using a simple repeater-to-repeater protocol, to one or more repeaters that are connected to distant LANs. The repeater on the receiving end will rebroadcast the packet on its LAN, retaining the original packet's source address. The broadcast repeater can be made very intelligent in its selection of messages to be forwarded. We currently have the repeater forward only broadcast messages sent using the UDP ports used by Cronus, but messages may be selected using any field in the UDP or IP headers, or all IP-level broadcast messages may be forwarded.

16.3. Alternatives to the Broadcast Repeater

We explored a few alternatives before deciding on our technique to forward broadcast messages. One of these methods was to put additional functions into the Internet gateways. Gateways could listen at the link-level for broadcast packets and relay the packets to one or more gateways on distant LANs. These gateways could then transmit the same packet onto their networks using the local network's link-level broadcast capability, if one is available. All gateways participating in this scheme would have to maintain tables of all other gateways which are to receive broadcasts. If the recipient gateway was serving a network without a capacity to broadcast it could forward the messages directly to one or more designated hosts on its network but, again, it would require that tables be kept in the gateway. Putting this sort of function into gateways was rejected for a number of reasons:

- it would require extensions to the gateway control protocol to allow updating the lists gateways would have to maintain;
- since not all messages (e.g., LAN address-resolution messages) need be forwarded, the need to control forwarding should be under the control of higher levels of the protocol than may be available to the gateways;
- Cronus could be put into environments where the gateways may be provided by alternative vendors who may not implement broadcast propagation;
- as a part of the underlying network, gateways are likely to be controlled by a different agency from that controlling the configuration of a Cronus system, adding bureaucratic complexity to reconfiguration.

Another idea which was rejected was to put broadcast functionality into the Cronus kernel. The Cronus kernel is a process which runs on each host participating in Cronus, and has the task of routing all messages passed between Cronus processes. The Cronus kernel is the only program in the Cronus system which directly uses broadcast capability (other parts of Cronus communicate using mechanisms provided by the kernel). We could either entirely remove the Cronus kernel's dependence on broadcast, or add a mechanism for emulating broadcast using serially-transmitted messages when the underlying network does not provide a broadcast facility itself. Either solution requires all Cronus kernel processes to know the addresses of all other participants in a Cronus system, which we view as an undesirable limit on configuration flexibility. Also, this solution would be Cronus-specific, while the broadcast-repeater solution is applicable to other broadcast-based protocols.

16.4. Implementation

The broadcast repeater is implemented as two separate processes - the forwarder and the repeater. The forwarder process waits for broadcast UDP packets to come across its local network which match one or more specific port numbers (or destination addresses). When such a packet is found, it is encapsulated in a forwarder-repeater message sent to a repeater process on a foreign network. The repeater then relays the forwarded packet onto its LAN using that network's link-level broadcast address in the packet's destination field, but preserving the source address from the original packet.

When the forwarder process starts for the first time it reads a configuration file. This file specifies the addresses of repeater processes, and selects which packets should be forwarded to each repeater process (different repeaters may select different sets of UDP packets). The forwarder attempts to establish a TCP connection to each repeater listed in the configuration file. If a TCP link to a repeater fails, the forwarder will periodically retry connecting to it. Non-repeater hosts may also be listed in the configuration file. For these hosts the forwarder will simply replace the destination broadcast address in the UDP packet with the host's address and send this new datagram directly to the non-repeater host.

If a repeater and a forwarder co-exist on the same LAN a problem may arise if the forwarder picks up packets which have been rebroadcast by the repeater. As a precaution against rebroadcast of forwarded packets (*feedback* or *ringing*), the forwarder does not connect to any repeaters listed in its configuration file which are on the same network as the forwarder itself. Also, to avoid a broadcast loop involving two LANs, each with a forwarder talking to a repeater on the other LAN, forwarders do not forward packets whose source address is not on the forwarder's LAN.

16.5. Experience

To date, the broadcast repeater has been implemented on the VAX running 4.2 BSD UNIX operating system with BBN's networking software and has proven to work quite well. Our current configuration includes two Ethernets which are physically separated by two other LANs. The broadcast repeater has successfully extended our broadcast domain to include both Ethernets even though messages between the two networks must pass through at least two gateways. We were forced to add a special capability to the BBN TCP/IP implementation which allows privileged processes to send out IP packets with another host's source address.

The repeater imposes a fair amount of overhead on the shared hosts that currently support it due to the necessity of waking the forwarder process on all UDP packets which arrive at the host, since the decision to reject a packet is made by user-level software, rather than in the network protocol drivers. One solution to this problem would be to implement the packet filtering in the system kernel (leaving the configuration management and rebroadcast mechanism in user code) as has been done by Stanford/CMU in a UNIX packet filter they have developed. As an alternative we are planning to rehost the implementation of the repeater function to a GCE. Such a machine is better suited to the task since scheduling overhead is much less than it is on a multi-user timesharing system.

REFERENCES

[BBN 5086]

Cronus, A Distributed Operating System: Interim Technical Report No. 1, R. Schantz, E. Burke, S. Geyer, M. Hoffman, A. Lake, K. Pogran, D. Tappan, R. Thomas, S. Toner, and W. MacGregor, Technical Report #5086, Bolt Beranek and Newman Inc., July 1982.

[BBN 5261]

Cronus, A Distributed Operating System: Interim Technical Report No. 2, R. Schantz, B. Woznick, G. Bono, E. Burke, S. Geyer, M. Hoffman, W. MacGregor, R. Sands, R. Thomas, and S. Toner Technical Report #5261, Bolt Beranek and Newman Inc., February 1983.

[BBN 5646]

Cronus, A Distributed Operating System: Interim Technical Report No. 3, M. Barrow, G. Bono, M. Dean, M. Hoffman, R. Sands, R. Schantz, R. Thomas and B. Woznick. Technical Report #5261, Bolt Beranek and Newman Inc., May 1984.

[BBN 5879]

Cronus, A Distributed Operating System: Functional Definition and System Concept, R. E. Schantz and R. H. Thomas, Technical Report #5879, Bolt Beranek and Newman Inc., RADC-TR-88-132, Vol II dated June 1988.

[BBN 5884]

Cronus System/Subsystem Specification. Technical Report #5884, BBN Laboratories, Inc., RADC-TR-88-132, Vol I dated June 1988.

[BBN 5885]

Cronus, A Distributed Operation System: Phase 1 Final Report (Interim Technical Report No. 4), R. Schantz, R. Thomas, R. Gurwitz, G. Bono, M. Dean, K. Lebowitz, K. Schroder, M. Barrow, and R. Sands. Technical Report #5885, Bolt Beranek and Newman Inc., January 1985.

[BBN 6073]

fIC2 System Internet Experiment Interim Technical Report No. 1, James C. Berets, Ronald A. Mucci, Richard E. Schantz and Kenneth J. Schroder. Technical Report #6073. BBN Laboratories Incorporated, October 1985.

[BBN 6180]

Cronus User's Manual. R. Sands and K. Schroder, eds. Technical Report #6180, BBN Laboratories, Inc., February 1986.

[BBN 6181]

Cronus Program Maintenance Manual. R. Sands and K. Schroder, eds. Technical Report #6181, BBN Laboratories, Inc., February 1986.

[BBN 6182]

Cronus Operator's Manual. R. Sands and K. Schroder, eds. Technical Report #6182, BBN Laboratories, Inc., February 1986.

BBN 6183

Cronus: A Distributed Operating System: Cronus DOS Implementation, Final Report; Interim Technical Report No. 6, R. Schantz, K. Schroder, M. Barrow, G. Bono, M. Dean, R. Gurwitz, K. Lam, K. Lebowitz, P. Neves, R. Sands. Technical Report #6183, BBN Laboratories, Inc., RADC-TR-88-132, Vol IV dated June 1988

Berets 1985

Cronus: A Testbed for Developing Distributed Systems, J. Berets, R. Mucci and R. Schantz. Proceedings of the 1985 IEEE Military Communications Conference, October 1985.

Dalal 1981

48-bit absolute internet and Ethernet host numbers, Yogen K. Dalal and Robert S. Printis, Proc. of the 7th Data Communications Symposium, October 1981.

DEC 1980

The Ethernet: a local area network, data link layer and physical layer specifications, Digital Equipment Corp., Intel Corp., and Xerox Corp., Version 1.0, September 1980.

Goldberg 1983

Smalltalk-80. The Language and Its Implementation, Adele Goldberg and David Robson, Addison-Wesley, Reading Ma. 1983.

Herlihy 1982

A Value Transmission Method for Abstract Data Types, M. Herlihy and B. Liskov, ACM Transactions on Programming Languages and Systems, Volume 4 (4) 527, October 1982.

Jones 1978

The Object Model: A Tool for Structuring Software, A. K. Jones, in *Operating Systems: An Advanced Course*, R. Bayer, R. M Graham, and G. Seegmuller, eds., Springer-Verlag, Heilelberg, 1978.

Liskov 1977

An Introduction to Formal Specifications of Data Abstractions, Barbara Liskov and Stephen Ziles, in *Current Trends in Programming Methodology*, Vol 1, Raymond T. Yeh, ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

NIC 1982

Internet protocol transition workbook, Network Information Center, SRI International, Menlo Park, California, March 1982.

Parker 1983

Detection of Mutual Inconsistency in Distributed Systems, D. S. Parker, Jr. et al, IEEE Transactions on Software Engineering, Volume SE-9 (3) 240, May 1983.

Postel 1981a

Assigned numbers, Jon Postel, RFC 790, USC Information Sciences Institute, September 1981.

Postel 1981b

Internet Protocol - DARPA internet program protocol specification, Jon Postel, ed., RFC 791,

USC/Information Sciences Institute, September 1981.

[Rentsch 1982]
Object oriented programming, T. Rentsch, SIGPLAN Notices, September 1982, pp. 51-57.

[Robinson 1977]
A Formal Methodology for the Design of Operating System Software, Lawrence Robinson, Karl N. Levitt, Peter G. Neumann, and Ashok R Saxena, in *Current Trends in Programming Methodology,* Vol 1, Raymond T. Yeh, ed., Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[Schantz 1986a]
The Architecture of the Cronus Distributed Operating System, R. Schantz, R. Thomas and G. Bono. Sixth Internaltional Conference on Distributed Computing Systems, May 1986.

[Schantz 1986b]
Programming Support in the Cronus Distributed Operating System, R. Schantz, M. Dean, and R. Gurwitz. Sixth International Conference on Distributed Computing Systems, May 1986.

[Weinreb 1981]
Lisp Machine Manual, Daniel Weinreb and David Moon, Massachusetts Institute of Technology, Cambridge Ma., 4th ed., 1981, p. 279f.

INDEX